Distributed Aspects: better separation of crosscutting concerns in distributed software systems

Luis Daniel Benavides Navarro

UNIVERSITÉ DE NANTES
UFR SCIENCES ET TECHNIQUES

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATHÉMATIQUES

Année 2009

N° attribué par la bibliotéque

Distributed Aspects:

better separation of crosscutting concerns in distributed software systems

Les aspects distribués : pour une meilleure séparation des préoccupations transverses dans les logiciels distribués

THÈSE DE DOCTORAT
Specialité: Informatique et Applications
Présentée
et soutenue publiquement par

Luis Daniel Benavides Navarro

Le 19 Janvier 2009 à Nantes, devant le jury ci-dessous

Président	:		
Rapporteurs	:	Shigeru CHIBA	Professeur, Tokyo Institute of Technology
		Lionel SEINTURIER	Professeur, Université de Lille
Examinateurs	:	Jean- Charles FABRE	Professeur, LAAS-CNRS
		Wouter JOOSEN	Professeur, Katholieke Universiteit Leuven
		Christian ATTIOGBÉ	Professeur, Université de Nantes
		Pierre COINTE	Professeur, École des Mines de Nantes
		Mario SÜDHOLT	Maître-Assistant, HDR, École des Mines de Nantes

Directeur de thèse : Pierre Cointe
Responsable scientifique : Mario Südholt

Laboratoire : UMR Laboratoire informatique de Nantes Atlantique (LINA)
Etablissement(s) d'accueil : École des Mines de Nantes-INRIA, LINA
Addresse : La Chantrerie – 4, rue Alfred Kastler – 44307 Nantes cedex 3 – France

ED:.....

Abstract

This thesis shows that abstractions provided by current mainstream Object Oriented (OO) languages are not enough to address the modularization of distributed and concurrent algorithms, protocols, or architectures. In particular, we show that code implementing concurrent and distributed algorithms is scattered and tangled in the main implementation of JBoss Cache, a real industrial middleware application. We also show that not only code is tangled, but also conceptual algorithms are hidden behind object-based structures (*i.e.*, they are not visible in the code). Additionally, we show that such code is resilient to modularization. Thus, we show that several cycles of re-engineering (we study the evolution of three different version of JBoss Cache) using the same set of OO abstractions do not improve on the modularization of distributed and concurrent code.

From these findings we propose a novel Aspect Oriented programming language with explicit support for distribution and concurrency (AWED). The language uses aspects as main abstractions and propose a model for distributed aspects and remote pointcuts, extending sequential approaches with support for regular sequences of distributed events. The language also proposes advanced support for the manipulation of groups of host, and the fine-grained deterministic ordering of distributed events. To evaluate the proposal we perform several experiments in different domains: refactoring and evolution of replicated caches, development of automatic toll systems, and debugging and testing of distributed applications.

Finally, using this general model for distribution we provide two additional contributions. First, we introduce Invasive Patterns, an extension to traditional communication patterns for distributed applications. Invasive Patterns present an aspect-based language to express protocols over distributed topologies considering different coordination strategies (Architectural programming). The implementation of this approach is leveraged by the distributed features of AWED and is realized by means of a transformation into it. Second, we add the deterministic manipulation of distributed messages to our model by means of causally ordered protocols.

Résumé

Cette thèse montre que les abstractions fournies pour les langages orientés objets (OO) ne sont pas suffisantes pour répondre aux problèmes de modularisation des algorithmes, protocoles, et architectures pour la distribution et la concurrence. Ainsi, le code dédie à la concurrence et à la distribution se trouve souvent dispersé et entrelacé dans l'implémentation des logiciels distribués. En plus, nous montrons que ce code est résistante à la modularisation et que même après plusieurs cycles de re–ingénierie, en utilisant le même paradigme de programmation OO, la modularisation du code pour la distribution et la concurrence ne s'améliore pas.

À partir de ces résultats, nous proposons un nouveau langage pour la programmation à aspects avec des abstractions explicites pour la distribution et la concurrence (AWED). Le langage propose un modèle pour les aspects distribués et les coupures distantes, en étendant des travaux sur les aspects séquentiels (non distribués) avec des mécanismes de langage pour la détection de séquences des événements distribués. Le langage propose également un support avancé pour la manipulation des groupes des hôtes, l'exécution distante d'actions, et le ordonnancement déterministe des messages distribués. Plusieurs expériences ont permis de valider notre travail dans différents domaines: la re–factorisation et l'évolution d'un intergiciel distribué, le développement de systèmes de péage automatique, et le débogage et les tests des applications distribuées.

Finalement, à l'aide de ce modèle général pour la distribution, nous proposons deux contributions supplémentaires. Tout d'abord, nous présentons les patrons invasifs: une extension des patrons de communication traditionnels pour les applications réparties. Les patrons invasifs présentent un langage d'aspects pour permettant d'exprimer des protocoles distribués sur différents topologies et avec différents stratégies de coordination. Deuxièmement, nous ajoutons à notre modèle la manipulation déterministe de messages distribués en utilisant les relations de causalité entre messages.

Mots clés: Aspects, langages de programmation, intergiciel distribué, motifs envahissantes, AWED.

Discipline: Informatique et applications.

N°:.....

Contents

Abstract iii

Résumé v

1 Introduction 1
- 1.1 Crosscutting concerns in the JBoss Cache distributed middleware . . . 2
- 1.2 Contributions . . . 3
- 1.3 Structure of the document . . . 5

2 State of the art 7
- 2.1 A taxonomy for distributed aspects . . . 7
  - 2.1.1 Communication model . . . 7
  - 2.1.2 Remote pointcut model . . . 8
  - 2.1.3 Remote advice model . . . 10
  - 2.1.4 Aspect model . . . 10
  - 2.1.5 Aspect composition . . . 11
  - 2.1.6 The complete taxonomy . . . 12
- 2.2 Sequential aspect languages and distribution . . . 12
  - 2.2.1 AspectJ languages' structure . . . 12
  - 2.2.2 History-based aspects . . . 16
  - 2.2.3 Distribution and concurrency using sequential AOP . . . 19
  - 2.2.4 Classification and discussion . . . 20
- 2.3 Aspect oriented programming for distributed applications . . . 22
  - 2.3.1 Domain specific languages . . . 22
  - 2.3.2 Frameworks for distributed AOP . . . 25
  - 2.3.3 Classification ad discussion . . . 31
- 2.4 Language support for distributed aspects . . . 31
  - 2.4.1 Remote pointcuts in DJCutter . . . 31
  - 2.4.2 Aspect scoping and instantiation . . . 34

3 Crosscutting and evolution of JBoss Cache 37
- 3.1 Overview of replication and transaction management . . . 38
- 3.2 JBoss Cache implementation: principles and crosscutting . . . 40
  - 3.2.1 Design principles . . . 40
  - 3.2.2 Implementation structure and crosscutting issues . . . 41
  - 3.2.3 Caching of POJOs and JBoss AOP. . . . 42
- 3.3 Evolution and crosscutting in JBoss Cache . . . 42

3.3.1 Evolution of the interceptor framework . . . 43
3.3.2 Implementation structures and crosscutting . . . 44
3.3.3 Quantitative analysis of crosscutting concerns . . . 46
3.4 Discussion . . . 48

4 The AWED language 49
4.1 AWED as a distributed aspect language . . . 49
4.2 Syntax and semantics . . . 51
4.2.1 Pointcuts . . . 51
4.2.2 Sequences . . . 54
4.2.3 Parameter passing . . . 55
4.2.4 Advice . . . 56
4.2.5 Aspects . . . 59
4.3 AWED by example . . . 59
4.3.1 Distribution . . . 59
4.3.2 Clustering . . . 60
4.3.3 Caching revisited . . . 61
4.4 Advanced examples: JBoss cache extension and refactoring . . . 64
4.4.1 Refactoring of JBoss replication code . . . 64
4.4.2 Extension of the JBoss replication strategy . . . 65
4.4.3 Comparison to a JBoss-only solution . . . 69
4.5 Implementation . . . 69
4.5.1 AWED implementation overview . . . 70
4.5.2 AWED architecture . . . 70
4.5.3 Remote pointcuts . . . 72
4.5.4 Aspect Distribution . . . 74
4.5.5 Asynchronous Advice and Futures . . . 75
4.5.6 State Sharing . . . 75
4.5.7 Optimizations . . . 76

5 Invasive Patterns 77
5.1 Introduction . . . 77
5.2 Pattern-based approaches for distributed development . . . 78
5.2.1 Massively parallel patterns . . . 78
5.2.2 Architectural patterns for distributed applications . . . 79
5.2.3 Design Patterns for distributed applications . . . 81
5.2.4 Aspect oriented pattern approaches . . . 81
5.3 Pattern-like structures in distributed middleware . . . 83
5.3.1 Pattern-like structures in JBoss Cache . . . 83
5.3.2 Source code representation of pattern-like structures . . . 85
5.4 Motivation and requirements for Invasive patterns . . . 85
5.5 Invasive patterns . . . 87
5.5.1 Structure and design . . . 87
5.5.2 Synchronization . . . 88
5.6 Pattern language . . . 89
5.6.1 Syntax and informal semantics . . . 90
5.7 Implementation . . . 91

5.7.1 Transformation of invasive patterns into AWED . . . 92
5.8 Evaluation . . . 94
5.8.1 JBoss Cache revisited . . . 95
5.8.2 Qualitative and quantitative evaluation . . . 101
5.9 Grids: a case of study for invasive patterns . . . 101
5.9.1 Motivation: communication patterns on grid applications . . . 102
5.9.2 Evaluation . . . 103
5.10 Discussion and future work . . . 108

6 Causally ordered sequences 111
6.1 Related work . . . 112
6.2 Motivation . . . 113
6.2.1 Expressive breakpoints for distributed debugging . . . 113
6.2.2 Test-driven development . . . 115
6.3 Language support . . . 115
6.3.1 Distributed debugging with AWED . . . 116
6.3.2 The case for causality relationships . . . 118
6.3.3 Background: logical time and causality . . . 119
6.3.4 AWED with causal pointcuts . . . 122
6.4 Implementation . . . 123
6.4.1 AWED architecture . . . 124
6.4.2 Adding causality to non-causal distributed applications . . . 126
6.5 Evaluation . . . 127
6.5.1 Qualitative evaluation . . . 127
6.5.2 Micro-Benchmarks . . . 129
6.5.3 Remote debugging vs. distributed debugging . . . 131
6.6 Discussion . . . 132

7 Conclusion and future work 133
7.1 Future work and perspectives . . . 134

List of Figures

1.1 Crosscutting concerns in JBoss Cache's interceptor package (version 2.0.0.GA) 3
1.2 A first example of tangled code in class Replication interceptor of JBoss Cache 2.0.0.GA 4

2.1 AspectJ's aspect example 14
2.2 Excerpt of AspectJ grammar defining basic pointcut constructs 15
2.3 COOL coordinator example 23
2.4 RIDL language (excerpts) 23
2.5 RIDL portal example 24
2.6 Aspect component for replication in JAC framework 28
2.7 Replication in DyMAC 30
2.8 Distributed deployment in CaesarJ 35

3.1 Architecture of transaction handling with replication in JBoss Cache 39
3.2 Interceptor chain pattern implementation in JBoss Cache. 40
3.3 Low-level transaction handling in class `TreeCache` 41
3.4 Code structure of JBoss Cache: a) Class diagram of filter pattern implementation, b) Crosscutting diagram of scattered and tangled code for distribution and transactions. 43
3.5 Evolution of code structure in JBoss Cache 46
3.6 Entangled and scattered code for distribution (gray) and transactions (black) in the evolution of JBoss Cache. On the left of each figure the main class implementing the tree data structure, and on the right side the interceptors package. 47

4.1 AWED language 52
4.2 Remote pointcuts and advice in AWED 53
4.3 Automaton representing first sequence example 55
4.4 Parameter passing example using a pointcut definition 56
4.5 Around advice chaining in AWED. Advice applicable to the same joinpoint execute in a single advice chain, regardless of execution host. 57
4.6 Simple authentication around advice 58
4.7 Distribution as an aspect 58
4.8 Distribution as an aspect 60
4.9 Clustering as an aspect 61
4.10 Cache replication as an aspect 61
4.11 Adaptive cache behavior 62

4.12 Aspect-based cooperative cache 63
4.13 Refactoring the JBoss replication code (principle) 65
4.14 Refactoring the JBoss replication code (detailed excerpt) 66
4.15 Extending the JBoss replication strategy 67
4.16 JBoss cache transaction replication implementation 68
4.17 Main components used in the implementation of AWED. 70
4.18 AWED architecture. 72
4.19 Detailed runtime behavior and architecture of the registry framework. 73
4.20 AWED's remote reference model implementation. The actual object is referenced by a remote proxy object and represented by a dummy object of the same type in the remote host. Such a dummy object is instrumented by an aspect that redirects calls to the remote proxy. The remote proxy then redirects calls to the actual object. 74
4.21 State sharing as a AWED aspect 76

5.1 H tree topology 79
5.2 Pipes and filter pattern 80
5.3 Standard startegy of filter pattern [AMC+03] 81
5.4 Sequence diagram of standard startegy of filter pattern [AMC+03] 82
5.5 Sequence diagram for custom filter pattern: implementation using Decorator pattern [GHJV94] 82
5.6 Architecture of transaction handling with replication in JBoss Cache 84
5.7 Tangled code of a two phase commit (2PC) protocol inside the invoke method of the `DataGravitationInterceptor` class. 86
5.8 Basic patterns 87
5.9 Invasive patterns 88
5.10 Pattern language 90
5.11 Pattern Compositions 90
5.12 A Session Profiling Aspect 91
5.13 Transformation into AWED 93
5.14 Pattern-based definition of the JBoss Cache two phase commit 95
5.15 Pointcut definition for transactional behavior in the pipe aspect 96
5.16 2PC invasive aspect `Aprepare` 97
5.17 2PC aspects to complete the protocol 98
5.18 2PC invasive AWED aspect for the creation of the transactional behavior . . 99
5.19 2PC AWED aspects for gathering the response 100
5.20 Patterns for NAS Grid 102
5.21 Topology and state machine representation of the protocol implementation for check pointing. 104
5.22 Checkpoint concern implemented using AWED 105
5.23 Farm-Gather topology with NAS language 106
5.24 Farm-Gather topology with pattern language 106
5.25 Impact of AWED implementation in NASGrid 107

6.1 Distributed Cflow graphical representation 117
6.2 Graphical representation of a start-action(s)-stop automaton 118
6.3 Graphical representation of the *happened before* relation. 119

6.4 Lamport's scalar logical clocks. 120
6.5 Graphical representation of the behavior of a distributed system implementing causality with vector clocks 121
6.6 AWED with causal pointcuts 122
6.7 AWED architecture. 125
6.8 Deadlock detection test case method 127
6.9 Pointcut for deadlock detection in a synchronous transactional cache. 128
6.10 Aspect ensuring the generation of the buggy behavior for deadlock detection. 128

List of Tables

2.1 Taxonomy for the communication model in distributed aspect-based systems 8
2.2 Taxonomy for remote pointcut models in distributed aspect-based systems . . 9
2.3 Taxonomy for remote advice models in distributed aspect-based systems . . . 10
2.4 Taxonomy for aspect models in distributed aspect-based systems 11
2.5 Taxonomy for aspect composition models in distributed aspect-based systems 11
2.6 Taxonomy for distributed aspect-based systems 13
2.7 Classification of sequential aspect-based systems for the implementation of distributed systems 21
2.8 Classification of D as an aspect-based systems for distribution 26
2.9 Classification of AOP frameworks for distribution: JAC, ReflexD, and DyMAC 32
2.10 Classification of DJCutter as an aspect-based language for distribution 33
2.11 Classification of CaesarJ as an aspect-based systems for distribution 36

3.1 List of interceptors classes in the three versions of JBoss Cache classified according to the main features 45
3.2 Metrics evolution in the refactoring process of JBoss Cache 47

4.1 Taxonomy for distributed aspect-based systems 50
4.2 Overview of AWED's features and their corresponding implementation techniques 71

6.1 Test results of 100.000 requests with respectively 20% and 80% writes 130
6.2 Debugging session without breakpoints (left half) and with a high-frequency breakpoint (right half). 131

Chapter 1

Introduction

During design and implementation software engineers translate functionalities, domain concepts, and properties of a system into units of abstraction of a specific programming language, *e.g.*, functions, procedures, classes, objects, components or packages. However, some properties and functionalities cannot be encapsulated into these units of abstraction, and instead, have to be coded involving many small fragments that are scattered over and tangled with one another as part of a large code base. Such functionalities are called *crosscutting concerns* and said to crosscut the implementation. Furthermore, changes in the specification of one of the properties will affect the implementation artifacts and the encapsulation units (*e.g.*, a function, procedure or objects) of (potentially many) others.

Crosscutting has been identified as a major problem in software engineering. Many functionalities have been identified as crosscutting concerns, including such diverse ones as performance optimization in graphical systems [KLM+97] and disk prefetching in operating systems [CK03]. Lopes [VL97] has studied distribution and concurrency as crosscutting concerns. She has shown, in particular, how these concerns lead to tangled code in the context of an application managing a book library. In particular she showed how, when dealing with concurrency, several programming strategies involving semaphores, monitors, guards, and inheritance produced different degrees of tangling depending on the support in the programming language for each synchronization construct. She has observed a similar situation for the communication concern. For example, in object oriented languages serialization and parameter passing often lead to tangled code because *e.g.*, classes have to be split into smaller ones and handled separately to achieve an optimized size when passing objects. Similarly, other research studied qualitatively [HK02] and quantitatively [GSF+05] identified design pattern implementation as a source of crosscutting code.

Distributed applications implemented using current object oriented mainstream languages suffer from crosscutting code. In 1997 Kiczales et al. introduced Aspect Oriented Programming [KLM+97] as a programming technique using new language abstractions to modularize such crosscutting concerns. Even though distribution and concurrency were identified early as crosscutting concerns and first addressed during initial work on aspects languages [VL97], current mainstream aspect languages do not provide explicit abstractions to deal with distribution or concurrency. Furthermore, few approaches have investigated the use of aspects for distributed programming, notable exceptions being [PSD+04, LJ06, NST04, TT06]. Instead, mainstream aspect languages rely on the basic mechanisms for distribution and concurrency provided by base languages (*e.g.*, monitors or Remote method Invocation in Java).

Although concurrent and distributed applications can be implemented this way, crosscutting due to concurrency and distribution concerns cannot be resolved using this implementation method [SLB02, KG02]. This is a major problem for AOP because many large-scale applications involve distribution and concurrency as central concerns.

To address this significant problem, we have explored the following main hypotheses:

- Scattered and tangled code found in the implementation of distributed applications can be adequately modularized using an aspect language with explicit support for concurrency and distribution.
- The corresponding non-sequential aspects provide a conceptual basis for the adoption and development of new general-purpose programming artifacts for distributed applications.

To confirm these hypotheses we have structured our research as follows: first we have studied the problem of crosscutting code due to concurrency and distribution in the evolution of an industrial middleware (JBoss Cache); based on this study, and on previous research on aspects for distribution we have designed an aspect language with explicit support for distribution (AWED); on top of this language, we have then built a language supporting a new notion of distributed patterns for heterogeneous distributed applications; finally, we have explored means to deal with the non-determinism inherent to distributed applications, by introducing causal aware constructs at the language level and using them in the context of debugging tools for distributed middleware.

In the following, we briefly present an example of crosscutting in concurrent and distributed code. We then present a detailed overview of contributions presented in this dissertation. Finally, we present the structure of this document.

1.1 Crosscutting concerns in the JBoss Cache distributed middleware

As an example of how non-modular implementations of major concerns pose problems for distributed software systems we now briefly discuss replications and transactions in JBoss Cache (a detailed discussion is presented in chapter 3).

Scattering

Consider the example of a replicated transactional cache, a component commonly used to support clustering in distributed applications. In such a middleware replication (*i.e.*, distributed replication) and transactional support are two major concerns. However, we have found that even the most frequently used replication and transactional code is scattered over the implementation of replicated caches.

Figure 1.1 represents the code structure of package *interceptors*, a major part of JBoss Cache 2.0.0.GA [JBo08b], a popular replicated cache [1]. In the figure, the boxes represent classes, the black lines correspond to code for transactions and the gray lines correspond

[1] In order to create this kind of figures we use manually coded aspects to match method calls of the dedicated API and an automatic tool to generate the graphical representation. The tool used to generate such figures is the AspectJ plugin for the Eclipse development framework [asp08].

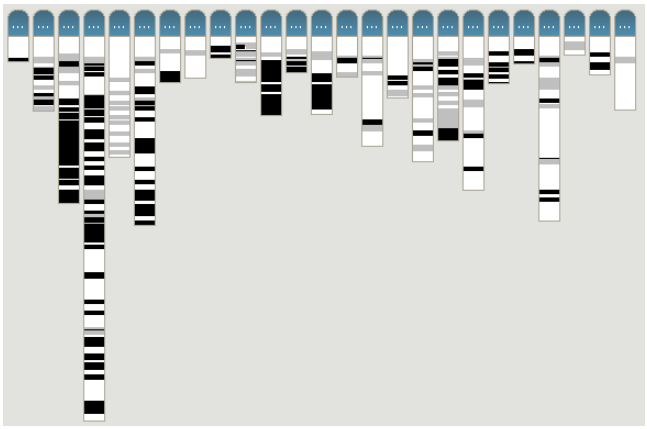


Figure 1.1: Crosscutting concerns in JBoss Cache's interceptor package (version 2.0.0.GA)

to code for replication. This figure shows that transactional and replication code is not encapsulated but is scattered over several classes. From the figure, we cannot evaluate the complexity of the scattered code. However, below, we show that scattering means in this case that statements get interlaced inside methods in a very fine-grained manner.

Tangling

Figure 1.2 shows an excerpt of code from the method *invoke* of class *ReplicationInterceptor* in JBoss Cache 2.0.0.GA [JBo08b]. The code includes statements from several functionalities: line 6 corresponds to code implementing a filter design pattern (see *Core J2EE patterns* [AMC+03]); line 7, lines 13–16, and lines 23–25 correspond to code for transactions; lines 9–11 correspond to code for replication; finally, lines 18–21 correspond to code for logging. In this method transactions, replication, filter management, and logging are thus tangled.

1.2 Contributions

Even though distribution and concurrency are major concerns in many current large-scale applications, current mainstream aspect-based approaches adopted a model for sequential AOP development. Such a model often proposes a mechanisms to query events in the execution of the application (pointcuts), a method-like mechanism to implement the concerns (advice), and a mechanism to define and bind pointcuts and advice. All these mechanisms only have local semantics, i.e., do not allow to refer, without recourse to the base language, to remote activities in any way.

The first steps towards a comprehensive model for distributed AOP have been proposed through the introduction of distributed aspects and remote advice in JAC [PSD+04], and the introduction of remote pointcuts as part of DJCutter [NST04]. This thesis builds on top of these approaches by presenting a much more expressive model for distributed AOP realized in form of two concrete aspect-based languages, and validated by several experiments over real-world distributed applications.

This thesis present contributions of different types. First, we have studied the problem of

```
1  public class ReplicationInterceptor extends BaseRpcInterceptor
2  {
3
4     public Object invoke(InvocationContext ctx) throws Throwable
5     {
6        MethodCall m = ctx.getMethodCall();
7        GlobalTransaction gtx = ctx.getGlobalTransaction();
8
9        // bypass for buddy group org metod calls.
10       if (MethodDeclarations.isBuddyGroupOrganisationMethod(m.getMethodId()))
11                      return super.invoke(ctx);
12
13       boolean isLocalCommitOrRollback =
14         gtx != null && !gtx.isRemote()
15         && (m.getMethodId() == MethodDeclarations.commitMethod_id
16         || m.getMethodId() == MethodDeclarations.rollbackMethod_id);
17
18       if (log.isTraceEnabled())
19            log.trace("isLocalCommitOrRollback? " +
20                      isLocalCommitOrRollback +
21                      "; gtx = " + gtx);
22
23       // pass up the chain if not a local commit or
24       // rollback (in which case replicate first)
25       Object o = isLocalCommitOrRollback ? null : super.invoke(ctx);
```

Figure 1.2: A first example of tangled code in class Replication interceptor of JBoss Cache 2.0.0.GA

crosscutting in detail by means of the analysis of the evolution of a large concrete distributed middleware. We have then proposed a general model for distributed AOP by means of the design of a concrete language supporting such a model. We have also implemented a compiler and runtime support for this language. Finally, we have validated our main hypotheses introduced above by addressing several different problems of distributed systems, and several experiments over real-world applications.

Concretely, this dissertation presents the following contributions:

- First, we analyze how the problem of scattered and tangled code, due to crosscutting concerns (in particular concurrency and distribution), applies to large-scale software systems. Furthermore, we analyze how the evolution of such software systems affects the modularization of crosscutting concerns. We have investigated, among others, the middlewares JBoss Cache [JBo08b] and Apache's ActiveMQ [sf08a], as well as the grid benchmarking application Nasgrid [Fru01].

- Based on the study of the evolution of crosscutting concerns in real-world industrial middleware, we designed an aspect oriented language with explicit distribution (AWED). This language provides as main features remote pointcuts, distributed control flow, pointcuts for regular sequences of remote events, a/synchronous distributed advice, distributed aspect deployment and instantiation, and group based communication.

- To validate our approach we have applied AWED to different problems of distributed

software systems, *e.g.*, implementing sophisticated replicated-caching policies, and refactoring and extension of JBoss Cache.

- We present a model to bridge the gap between high level architectures and implementation, by means a new notion of patterns, so called invasive distributed patterns.
- Finally, this dissertation presents language constructs for taking into account fine-grained message ordering and its application to the implementation of development tools for distributed applications. Concretely, we show that the implementation of new development tools (*e.g.*, debuggers) can benefit from research on determinism and correctness trough causal message ordering.

1.3 Structure of the document

This document consists, apart from the introduction, of six chapters.

- **Chapter 2** introduces a taxonomy for distributed aspect languages based on the taxonomy for distributed languages and calculi proposed by Caromel and Henrio [CH05], and providing an extension of taxonomy for aspect languages proposed by Südholt [Sü07]. Using this new taxonomy, the chapter presents a detailed analysis of aspect-based languages and frameworks for distribution.
- **Chapter 3** gives concrete evidence for the importance of crosscutting in middleware. Concretely, it presents the evolution of crosscutting concerns trough three versions of JBoss Cache. This study shows that even after several cycles of re-engineering the crosscutting concerns are resilient to modularization.
- **Chapter 4** presents AWED, an aspect language with explicit distribution. The chapter presents the basic assumptions and hypotheses taken during the design of AWED, the syntax and informal semantics of the language, and several examples to evaluate the applicability of the approach. The chapter also discuss the main implementation issues.
- **Chapter 5** introduces invasive patterns, a new notion of pattern for heterogeneous distributed applications and a corresponding pattern language. We show how these patterns can be used to refactor JBoss Cache and to completely modularize its replication and transaction functionalities by substantially reducing the complexity of its implementation at the same time. The chapter also presents a definition of invasive patterns using a formal transformation into AWED.
- **Chapter 6** considers ordering of messages in the distribution model of AWED. We propose, in particular, a language extension to support causal ordering of messages, allowing to restrict non-deterministic message orders to deterministic subtraces based on guarded state finite machines. We evaluate the resulting extension of AWED in the context of debugging scenarios for distributed middleware.
- **Chapter 7** concludes and presents directions of future work.

Chapter 2

State of the art

This chapter presents the state-of-the-art of Aspect Oriented approaches for distributed programming. The chapter first introduces the main concepts in the field by means of a taxonomy of AO concepts relevant for distribution approaches. Then we discuss the main approaches for aspects for distributed applications in detail, classifying them according to the taxonomy.

Note that this chapter only presents the approaches that are directly related to its main subject area, aspects and distribution. Related work on adjacent fields that is relevant to this thesis is presented in the chapters presenting corresponding contributions: related work on software patterns and aspects is presented in chapter 5 where invasive patterns are introduced; related work on causal ordering in distributed middleware and aspects is included in chapter 6 where our approach of causal aspects is detailed.

2.1 A taxonomy for distributed aspects

Distributed aspect languages and systems deal with the modularization of crosscutting concerns in distributed applications. However, each of the systems currently proposed provides different mechanisms to fulfil this objective. This section introduces a taxonomy that serves to characterize, differentiate, and compare distributed aspect-based systems. The taxonomy has been defined based on a study of distributed object languages by Caromel and Henrio (Chapter 2 of [CH05]), and a taxonomy of sequential aspects by Südholt [Sü07].

We have divided the taxonomy in four subsections. The communication and synchronization model is discussed in subsection 2.1.1. The remote pointcut model is discussed in 2.1.2. Subsection 2.1.3 discusses the remote advice model. Finally, subsection 2.1.5 presents the elements to characterize composition of distributed aspects.

2.1.1 Communication model

The first section of the taxonomy characterize the communication model (see table 2.1). Aspect-based languages and systems for distribution need communication capabilities that are either supported by the base language they extend, or by explicit mechanisms in the aspect language.

The first element in the communication model is the communication mechanism. A sequential (non-distributed) aspect system extending a language like Java often uses Remote Method invocation as its communication mechanism (see AspectJ [KHH+01]). A language

Taxonomy elements	Values
Communication model	
Communication mechanisms	
Remote method call	yes/no
Join point propagation	yes/no
Controlled remote advice invocation	yes/no
Group communication	yes/no
Parameter passing modes	by reference, or by copy
Synchronization model	
Synchronization mechanisms	synchronous control, non-blocking, or futures
Communication timing	synchronous, asynchronous with rendezvous, asynchronous FIFO, asynchronous without guarantee
Asynchronous hypothesis	yes/no
Causal predicates	yes/no

Table 2.1: Taxonomy for the communication model in distributed aspect-based systems

like AWED provides join point propagation and controlled remote advice invocation as communication (and distribution) mechanisms. Join point propagation refers to join points being sent as messages to be processed by remote aspects. Remote advice invocation refers to chains of applicable advice being controlled by the host where the join point occurs (see chapter 4 for a detailed explanation). Other systems, like DYMAC [LJ06] extend communication capabilities of component frameworks, like J2EE, by aspect support.

Once a communication mechanism is in place several other elements enter into consideration. First, passing values between remote processes are defined by parameter passing modes (*e.g.*, passing parameters by reference or by copy). Then, synchronization between processes needs to be expressible. Several mechanisms have been proposed for synchronization: an aspect system can, once again, rely on the base language mechanism (*e.g.*, JAVA monitors) or on explicit mechanisms introduced at the aspect level like AWED's futures.

Finally, the model defines the guarantees provided for the distribution of messages. For example, a system may be synchronous when a process is suspended waiting for the response of a remote process. It may also be asynchronous with rendezvous when the message is sent and a confirmation of delivery is received (the sender not blocking while the response is calculated). A system may also be asynchronous but preserve the order of messages coming from each process. Finally, a system may be asynchronous without any guarantee, but may provide additional control over messages, like AWED's causal predicates. Thus, instead of defining a total order on the distributed messages (all hosts seeing messages in the same order), a partial order is defined that may imply additional properties, *e.g.*, causality.

2.1.2 Remote pointcut model

Pointcut models are an essential element of aspect languages. A pointcut model defines the set of join points that can be considered, and the mechanisms to predicate over them (pointcuts). Join points are events occurring during the execution of a program that can be matched using pointcut expressions; such points are used to trigger specific behavior defined in pieces of advice (see below: advice models). Pointcuts are used to concisely define relations among (potentially many) join points. Pointcut expressions are constructs that match specific sets of join points. These expressions may be constructed using pointcut languages (*e.g.*, see systems

Taxonomy elements	Values
Remote pointcut model	
Expressiveness:	
Atomic	yes/no
Sequential control flow	yes/no
Distributed control flow	yes/no
History based	yes/no
Finite-state	yes/no
Vpa	yes/no
Context-free	yes/no
Turing-complete	yes/no
Remote join point support	
All pointcuts	yes/no
Location	Single host, Multiple host groups
Paradigm:	
Object-oriented	yes/no
Functional	yes/no
Logic	yes/no

Table 2.2: Taxonomy for remote pointcut models in distributed aspect-based systems

like AWED, DJCutter [NST04], or DYMAC [LJ06]) or API extensions (*e.g.*, see systems like ReflexD [TT06], or JAC [PSD+04]). Most of the aspect-based approaches proposing explicit distribution consider either pointcuts that can predicate over the localization of distributed join points, or local join points attached to remote pieces of behavior. Table 2.2 shows the part of our taxonomy concerning *remote pointcuts* of distributed.

A first major issue consists in the expressiveness of the model. The expressiveness ranges from atomic pointcuts that match sets of otherwise unrelated join points (*e.g.*, method calls), via pointcut models able to predicate over control flow relations (*e.g.*, a method call in the control flow of another method call, possibly a remote call), to history-based pointcut models. History-based pointcut models define pointcuts that enable relations to be expressed among several join points that occur in the execution of a program. Such relations can be expressed using languages of different expressiveness [1], including finite state automata, visibly pushdown automata [2] (VPA), and Turing-complete languages [3]. Finally, the hierarchy includes remote join points to describe systems allowing predicates over join points that occur on different locations.

The taxonomy also permits to classify the different approaches according to the paradigm and the generality of their models. Pointcut models may use object-oriented abstractions, functional-languages abstractions, or logic-language abstractions to predicate over join points. Pointcut models may also be dedicated to a specific domains (domain specific pointcut languages), or be general purposes.

[1] See the Chomsky hierarchy [Cho59, Cho56] for a detailed explanation on grammar classification.

[2] See Alur and Madhusudan [AM04] for a detailed discussion on visibly pushdown automata.

[3] See Turing original paper "On Computable Numbers with an Application to the Entscheidungs Problem" [Tur36] for a detailed discussion on Turing machines and Chomsky hierarchy [Cho59, Cho56] for their use in grammar classification

Taxonomy elements	Values
Remote advice model	
Filtering and ordering	yes/no
Location	Single host, Multiple host groups
Synchronization mode	Synchronous and asynchronous
Parameter passing modes	by reference and by copy
Proceed support	yes/no
Remote host only	yes/no
Remote and originating host	yes/no
Synchronization mechanisms	Blocking, transparent futures
Mobility	Strong, weak
Reflective access to program state	yes/no

Table 2.3: Taxonomy for remote advice models in distributed aspect-based systems

2.1.3 Remote advice model

The advice model describes how pieces of advice (method-like elements) are defined and bound to specific pointcut expressions. The defining hierarchy in the taxonomy regarding the advice model is structured as shown in table 2.3.

Regarding distribution, an advice may be executed locally or remotely. When executed remotely the advice can be executed on one or several remote places, raising the problem of execution ordering, and filtering/selecting the locations where remote processes advice is to be executed. Additionally, such remotely executed pieces of advice may be executed synchronously or asynchronously with respect to the process that triggered the advice execution(s). The advice model must deal with the problem of data passing and coordination between processes. In particular, the system must define a model for parameter passing to the remote advice (that may be different than the parameter passing in the communication model before). The advice model may also include means for the coordination of concurrent processes among advice and the base application, for example, by means of futures [RHH85]. The taxonomy also also allows advice execution to be classified as using strong mobility (passing of objects and their execution state) and weak mobility (passing only program or object definitions) as classifying element. Finally, the taxonomy includes access to information from the base program via reflection as a classification element.

2.1.4 Aspect model

Aspects are often modeled as syntactic class-like units that define pointcuts, advice, methods, and local state (fields for OO approaches). The defining part of the taxonomy for aspects is structured as shown in table 2.4.

The first element in the hierarchy deals with the instantiation of aspects. Instantiation may be defined by static declarations, *e.g.*, aspects may define one instance per process, one instance per class, one instance per object, or one instance per control flow. Alternatively, this instantiation may also be determined dynamically at execution time. The model also discuss the deployment mechanism and the deployment scope (where aspects are deployed). Additionally, state sharing between aspect instances is an important issue. This sharing may be between aspects in the same process, aspects in processes belonging to the same group,

Taxonomy elements	Values
Aspect model	
Instantiation	Declarative, imperative
Declarative	
Per thread	yes/no
Per class	yes/no
Per object	yes/no
singleton	yes/no
Per cflow	yes/no
Per binding	yes/no
Deployment	Dynamic, static, imperative
Deployment scope	global,local, group of hosts, remote host
State sharing	global, local, group, no
Weaving mechanisms	static, load-time, framework-based

Table 2.4: Taxonomy for aspect models in distributed aspect-based systems

or aspects deployed in any process of the distributed application. Finally, the aspect model proposes aspect weaving mechanism as a classifying issue.

Taxonomy elements	Values
Aspect composition	
Type	Implicit: Non-deterministic, Deterministic, Undefined; Explicit
Mechanisms	Precedence: partial, total; Operator; Program
Object	Advice, aspect
Scope	All, stateful
Expressiveness	Finite-state, Turing-complete
Distributed guarantee	yes/no

Table 2.5: Taxonomy for aspect composition models in distributed aspect-based systems

2.1.5 Aspect composition

The final part of the taxonomy deals with aspect composition. Two aspects that may match the same join point in the executing application, the means provided to deal with this interaction are defined by the composition model. Note that this is a limited notion of composition and more general models are conceivable but have not (yet) been included in models for distributed aspects. The composition taxonomy shown in table 2.5.

The first classification element in the taxonomy is the type of the composition. Most aspect-based languages provide implicit ways to deal with interactions. For example, by defining the order in which the weaver weaves aspects may determine how they are composed. The model may also provide means to explicitly define compositions or leave the conflict unresolved. For example, AspectJ provides a mechanism to define precedence but if precedence is not declared the application order of interacting advice is undefined.

The taxonomy also proposes the composition mechanisms themselves as classification element. Several mechanisms may be used to deal with composition in aspect-based approaches. AspectJ proposes a precedence language. EAOP [DFS05] provides operators to compose as-

pects. JASCO [SVJ03] provides program hooks to deal with composition at the program level. DyMAC proposes aspect compositions in the application description over a component framework (a la J2EE).

Additionally, these mechanisms may be applied over different elements like aspects or pieces of advice (see category Object), they may also have different scopes (see category scope). The scope of the composition may be the whole execution of the program or be defined according to the current state of the program (stateful). The composition approaches can also be classified according to the expressiveness of the composition mechanism, ranging from approaches comparable to finite-state languages to Turing-complete approaches.

Finally, the taxonomy includes an element related to the coherent control of composition over aspects running in multiple sites. AWED implements such control in its model.

2.1.6 The complete taxonomy

Table 2.6 shows the complete taxonomy.

2.2 Manipulation of distributed infrastructures using sequential aspect languages

One of the main research question of Aspect Oriented Programming (AOP) [KLM+97] is what language constructs allow programmers to well modularize crosscutting concerns. Several proposals have been presented to address this question, even before the term AOP was introduced (see for example composition filters [AWB+94], and work on meta-object protocols and reflection [KdRB91, MWY91, Chi95, KFRGC98]). In 1997 Kiczales at al. proposed Aspect Oriented programming to address encapsulation of crosscutting concerns [KLM+97] using aspect oriented languages. In general, an AOP language is defined on top of a base language (*e.g.*, Java [GJSB05]) used to express the actual application, and an advice language to express the crosscutting concern, and a pointcut language to compose and to coordinate base functionality and crosscutting concerns. These ideas have been embodied in AspectJ [KHH+01] in 2001.

This section first introduces the basic mechanisms of aspect oriented programming by means of an analysis of AspectJ, then briefly introduces more advanced concepts of other sequential aspect languages, and then considers their application to distributed infrastructures.

2.2.1 AspectJ languages' structure

AspectJ is a general purpose aspect oriented language that extends the Java programming language. AspectJ introduces aspects as syntactic units to modularize crosscutting concerns. An aspect is basically composed of pointcut definitions, defining points of interest in the base program, and pieces of advice. Pointcuts are defined using a pointcut language that semantically denotes sets of execution events (*i.e.*, join points). The pieces of advice bound to such pointcut definitions are method-like constructs that define what to do when one of the points of interest is reached. An advice can, in particular, execute some code before, after or instead of a join point. We now present in some detail these basic mechanisms. We do not consider advanced mechanisms here, such as aspect instances and application of aspects to other aspects: these will be introduced later as needed.

Taxonomy elements	Values
Communication model	
Communication mechanisms	
Remote method call	yes/no
Join point propagation	yes/no
Controlled remote advice invocation	yes/no
Group communication	yes/no
Parameter passing modes	by reference and by copy
Synchronization model	
Synchronization mechanisms	synchronous control, non-blocking, or futures
Communication timing	synchronous, asynchronous with rendezvous, asynchronous FIFO, asynchronous without guarantee
Asynchronous hypothesis	yes/no
Causal predicates	yes/no
Remote pointcut model	
Expressiveness:	
Atomic	yes/no
Sequential control flow	yes/no
Distributed control flow	yes/no
History based	yes/no
Finite-state	yes/no
Vpa	yes/no
Context-free	yes/no
Turing-complete	yes/no
Remote join point support	
All pointcuts	yes/no
Location	Single host, Multiple host groups
Paradigm:	
Object-oriented	yes/no
Functional	yes/no
Logic	yes/no
Remote advice model	
Filtering and ordering	yes/no
Location	Single host, Multiple host groups
Synchronization mode	Synchronous and asynchronous
Parameter passing modes	by reference and by copy
Parameter passing mechanism	tag language, object graph query language
Proceed support	yes/no
Remote host only	yes/no
Remote and originating host	yes/no
Synchronization mechanisms	Blocking, transparent futures
Mobility	Strong, weak
Reflective access to program state	yes/no
Aspect model	
Instantiation	Declarative, imperative
Declarative	
Per thread	yes/no
Per class	yes/no
Per object	yes/no
singleton	yes/no
Per cflow	yes/no
Per binding	yes/no
Deployment	Dynamic, static, imperative
Deployment scope	global,local, group of hosts, remote host
State sharing	global, local, group, no
Weaving mechanisms	static, load-time, framework-based
Aspect composition	
Type	Implicit: Non-deterministic, Deterministic, Undefined; Explicit
Mechanisms	Precedence: partial, total; Operator; Program
Object	Advice, aspect
Scope	All, stateful
Expressiveness	Finite-state, Turing-complete
Distributed guarantee	yes/no

Table 2.6: Taxonomy for distributed aspect-based systems

Aspects

As mentioned before the aspect is the main unit of encapsulation in AspectJ. An aspect is a class-like construct that is composed of method, variable, pointcut and advice declarations. Figure 2.1 shows a simple aspect. Aspects, as Java classes, may have package and import declarations, the figure illustrates that the aspect belongs to package `edu.emn.awed`, see line 1. Line 3 shows the declaration of the aspect `InvokeAspect` with the access modifier `public`. In the aspect body, a protected integer variable `i` is declared (see line 5). Between lines 7 and 9, the public method `increaseCounter` is defined. A pointcuts definitions is defined in lines 11 to 12. The keyword `pointcut` starts the definition of the `invokeCall` pointcut with an `Object` argument. After the colon a pointcut expression is defined. Here,

```
1  package edu.emn.awed;
2
3  public aspect InvokeAspect {
4
5      int i = 0;
6
7     private void increaseCounter(){
8         this.i++;
9     }
10
11     pointcut invokeCall(Object o):
12           call(* org.jboss.cache.interceptors.*.invoke(..)) && args(o);
13
14     before(Object o): invokeCall(o){
15         increaseCounter();
16     }
17 }
```

Figure 2.1: AspectJ's aspect example

the expression defines a pointcut that matches all the calls to method `invoke` in any class of package `interceptors`. Note that the parameter of the `call` pointcut is a method pattern, and the pointcut matches each method call whose signature matches that method pattern. Finally, lines 14 to 16 show an advice definition. Here, the advice increases the counter before each call matching the pointcut definition.

Pointcut language

Pointcuts are language constructs that typically quantify over different execution events. They may be defined using various mechanisms, frequently in terms of sets or logical expressions that match join points [GB03]. The pointcut language of AspectJ [KHH+01] quantifies over specific statically defined points of the program structure *e.g.*, method calls, and based on some dynamic conditions, *e.g.*, a method call occurring within the control flow of another method call.

Figure 2.2 shows an excerpt of AspectJ grammar defining basic pointcut constructors. AspectJ proposes a model that includes pointcuts to match method calls, method executions, object initialization, field referencing and setting, and advice execution. Each of these join points have associated information about the executing object (extracted using the `this` pointcut), the target object (extracted using the `target` pointcut), and, in the case of methods, the arguments (`args` pointcut). Additionally, the model proposes pointcuts that relate pairs of join points, thus these pointcuts are triggered when a relation between two pointcuts is fulfilled. In particular, AspectJ the model proposes pointcuts to match join points in the control flow of other join points (see pointcuts `cflow` and `cflowbelow`), and pointcuts to match join points inside methods, constructors, and class definitions (see pointcuts `within` and `withincode`). Finally, the model proposes an `if` pointcut that is parameterized with a boolean condition, and pointcuts predicating over annotations (not further described here). A detailed description of this language is outside the scope of this thesis, the interested reader may read the documentation found in [asp08].

The following excerpt of AspectJ grammar defines pointcut expressions:

$$
\begin{array}{rcl}
Pc & ::= & \texttt{call}(MethodOrConstructorPattern) \\
& | & \texttt{execution}(MethodOrConstructorPattern) \\
& | & \texttt{get}(FieldPattern) \\
& | & \texttt{set}(FieldPattern) \\
& | & \texttt{handler}(TypePattern) \\
& | & \texttt{initialization}(ConstructorPattern) \\
& | & \texttt{preinitialization}(ConstructorPattern) \\
& | & \texttt{staticinitialization}(TypePattern) \\
& | & \texttt{adviceexecution}(\ ) \\
& | & \texttt{this}(TypeOrIdentifier) \\
& | & \texttt{target}(TypeOrIdentifier) \\
& | & \texttt{args}(FormalsOrIdentifiersPattern) \\
& | & \texttt{cflow}(PcExpression) \\
& | & \texttt{cflowbelow}(PcExpression) \\
& | & \texttt{within}(TypePattern) \\
& | & \texttt{withincode}(MethodOrConstructorPattern) \\
& | & \texttt{if}(BooleanJavaExpression) \\
& | & AnnotationPointcut
\end{array}
$$

Figure 2.2: Excerpt of AspectJ grammar defining basic pointcut constructs

$$
\begin{array}{rcl}
PcExpression & ::= & Pc \\
& | & !PcExpression \\
& | & (PcExpression) \\
& | & PcExpression \ \&\& \ PcExpression \\
& | & PcExpression \ || \ PcExpression,
\end{array}
$$

A primitive pointcut (Pc) defines a set of join points in the execution of a program. Programmers can then construct expressions using union (||), intersection (&&), and complement (!).

For example, a method call join point can be matched using a `call` primitive pointcut:

call(Object edu.emn.Client.invoke(String))

This pointcut definition matches the calls to method `invoke` in class `Client` in the package `edu.emn`. A method call is only picked out if the method signature matches exactly the method pattern definition. In the previous example, the `call` pointcut only matches the method call if it returns an object of type `Object` and has one argument of type `String`. We can restrict the set of join points, *e.g.*, , as follows:

call(Object edu.emn.Client.invoke(String)) && **this**(edu.emn.Controller)

This pointcut matches only the calls to method `invoke` that are made by an objects of static type `Controller`.

The following is an example of the `cflow` pointcut:

call(Object edu.emn.Client.invoke(String)) &&
cflow(**call**(∗ edu.emn.Controller∗(..))),

picks out the calls to the method `invoke` that occur in the control flow, *i.e.*, between the entry and exit of calls to methods of an object of static type `Controller`.

Advice

AspectJ advice is a method-like construct that is triggered when a join point is matched by an associated pointcut definition. AspectJ advice allows advice to be executed before, after or instead of a specific joinpoint. The proceed pseudo-method is used to call the matched join point (in case it is a method call or execution) form advice. Furthermore, reflective capabilities can be used to access specific information relative to the join point and the aspect application. For example, a programmer defining an advice triggered by a method call may access the signature of the method call and change its target object and its arguments.

A simple example of an advice definition is the following:

```
Object around(Object o): invokeCall(o){
    Class callerObjectType = thisJoinPoint.getThis().getClass();
    Class targetObjectType = thisJoinPoint.getTarget().getClass();
    increaseCounter(callerObjectType, targetObjectType);
    return proceed(o);
}
```

The code shows an around advice (*i.e.*, executed instead of the triggering call point) that returns an object. The two first lines in the body of the advice use the `thisJoinPoint` object to access the reflective information of the current join point, in this case to extract the information about the caller object and the target object. Then the advice calls the method `increaseCounter` that counts the calls between objects of different types (see parameters `callerObjectType` and `targetObjectType`). Finally, the method calls the original behavior of the current joinpoint using the `proceed` keyword and returns the corresponding value.

2.2.2 History-based aspects

AspectJ-like languages propose a set of atomic pointcuts (see [Süd07]), *i.e.*, pointcuts that match individual unrelated events in the program execution. Such pointcuts do not permit to predicate over the history of events (In AspectJ the `cflow`, and in some sense, the `if` pointcuts are the only exceptions). In contrast, Douence et al. [DFS02, DFS05], Walker and Viggers [WV04], and Allan et al. [A+05], among others, have explored language features to predicate over event relations. These kind of stateful relations are particularly useful to handle crosscutting in heterogeneous distributed algorithms as shown in chapters 5 and 6 of this document. This section briefly presents the main characteristics of such history–based aspect languages by first introducing one of the first such approaches, *regular aspects*, and then a more recent, influential one,*tracematches*. Other work in the area has addressed non-regular, non turing-complete pointcut languages [NS06, WV04]. For example, Nguyen and Südholt propose an aspect language based on visibly pushdown automata. However, since our work is not based on such explicit representation of non–regular aspects, we do not consider these approaches any further.

Regular aspects

In 2002, Douence, Fradet, and Südholt [DFS02, DFS05] introduced stateful aspects as part of a framework for the resolution of aspects interactions. This model was based on a form of regular expressions between execution events to be used as triggering condition of advice. Concretely, this model abstracts the base program into a sequence of events, and the weaver is defined as a monitor that applies each aspect to appropriate events of the base execution.

An example of an aspect language instantiating such model is defined using the following grammar (presented by Douence et al. in [DFS05]):

$$\begin{array}{rcll} A & ::= & C \triangleright I; A & ;\textit{sequence} \\ & | & C \triangleright I; \textit{var} & ;\textit{end of sequence} \\ & | & A_1 \,\square\, A_2 & ;\textit{choice} \end{array}$$

The grammar allows recursive definition of aspects, sequence composition of aspects, and aspect composition by deterministic choice. The basic aspect sequence is defined by the form $C \triangleright I; A$, where C stands for an atomic pointcut, I (*inserts*) stands for an advice, and A is the next aspect in the sequence. The form $C \triangleright I; \textit{var}$ allows the basic case for recursion and marks the end of a sequence. Finally, $A_1 \,\square\, A_2$ presents a prioritized choice operator: A_2 is applied only if A_1 cannot be applied.

As a concrete example consider a replication aspect for a group of hosts:

$\textit{RplStart} =$ `StartRpl(group)` $\triangleright$ `skip` ; $\textit{RplAsp}$
$\textit{ReplAsp} =$ (`StopRpl(group)` $\triangleright$ `skip` ; $\textit{RplStart}$) $\square$
((`putValue(x)` $\triangleright$ `rManager.put(x,group)` $\square$ `getValue(x)` $\triangleright$ `skip`); $\textit{RplAsp}$)

The aspect *RplStart* defines an aspect sequence that waits for the method call `StartRpl` to start replicating in a specific group of hosts, the group name is bound to the free variable *group*. The advice `skip` defines an action that does nothing. The aspect *ReplAsp* defines a choice composition: the first branch waits for a join point stopping replication behavior on the specific group. If this branch is not matched, the right branch waits for a *put* (event put value) or *get* (event get value) method calls. If a *put* value method call is matched, the value is replicated to the specific group, calls to *get* value are ignored.

In this Ph.D. work we apply and extend this model in a distributed setting. We provide, in particular, different forms of distributed sequence pointcuts that permit to predicate over regular sequences of distributed events. Furthermore, an `if` pointcut allows the matching of turing–complete protocols.

Tracematches

Alan et al. [A$^+$05] have presented another aspect-based language feature, called *tracematch*, to match regular patterns of events in a program execution. This construct provides more extensive support for free variables than *regular aspects* [DFS05]. In particular, tracematches allow events to be matched not only based on the event kind but also on the values associated to the free variables. A tracematch is defined as follows:

$$\begin{array}{rcl} \textit{TraceMatch} & ::= & [\texttt{perthread}]\ \texttt{tracematch}(\textit{VarDecl}) \\ & & \{ \\ & & \textit{TokenDcl} + \\ & & \textit{REGEX} \\ & & \textit{MethodBody} \\ & & \} \end{array}$$

In the tracematch declaration the `perthread` keyword indicates if the matching is done over a particular thread (the default behavior matches events over the entire application). The `tracematch` keyword indicates that a tracematch is being defined with a list of free variables

(non terminal *VarDecl*). In the body of the tracematch three components are defined: the symbols of the regular expression (This is in contrast to *regular aspects* and allows to forbid matching of events that are not declared as symbols), a regular expression, and the method that will be triggered when the final symbol in the regular expression is matched. Symbols are defined as follows:

$$\textit{TokenDcl} \ ::= \ \texttt{sym}\ \textit{Name}\ \textit{Kind}\ :\ \textit{Pointcut};$$

The symbols are named (non terminal *Name*) and then defined using pointcuts, *e.g.*, a method call, and a reference point for the pointcut. A reference point may refer to the point before, or after a method call, or even after a method return or throwing an exception.

The following example shows how cache replication can be supported using tracematches.

```
tracematch (Group g, Cache c, Value v){
  sym start_replication after:
          call(* Cache.start(Group)) && args(g) && target(c);

  sym replicate_value after:
          call(* Cache.put(Value)) && args(v) && target(c);

  sym stop_replication after:
          call(* Cache.stop(Group)) && args(g) && target(c);

  start_replication replicate_value*
  {
     replicationManager.put(g, v);
     logManager.log(c,g,v); // replication action logged
  }
}
```

The example defines in its header three variables: `g` of type `Group`, `c` of type `Cache`, and `v` of type `Value`. Three symbols are defined. First, symbol `start_replication` matches all the calls to the method `start`, on objects of type `Observer`, with an argument of type `Group` as parameter. The pointcut `args(g)` binds the value of the argument to `g`, and the pointcut `target(c)` binds the target object of type `cache` to `c`. The second symbol, `replicate_value`, matches all the calls to the method `put` on the object of type `Cache`. Because of the use of the pointcut `target(c)` the call is only matched if the value corresponds to the previously bound value. The third symbol matches all the calls to method `stop`. This pointcut only matches if the target object of type `Cache` and the parameter of type `Group` are the same as those bound to `c` and `g`.

Finally the tracematch defines the regular expression and the advice to be applied when the last symbol of the regular expression is matched. The regular expression matches one symbol `start_replication` and zero or more symbols `replicate_value`. The symbol `stop_replication` is part of the alphabet but not of the regular expression definition. This symbol serves to stop the matching of the regular expression.

An interesting point of this approach is that a single tracematch definition can handle replication of multiple `cache` objects over different groups of replication, supporting even dynamic group creation. However, the advice is bound only to the last symbol on the regular expression, limiting the expressivity and power of the defined aspects. Furthermore, the pointcuts match only local events.

2.2.3 Distribution and concurrency using sequential AOP

Distribution and concurrency have been early identified as crosscutting concerns. However, it has been shown that sequential AOP, in particular AspectJ, is not sufficient to achieve the modularization of such concerns [SLB02, KG02].

Kienzle and Guerraoui [KG02] studied the implementation, using aspects, of concurrency and fault safety in distributed systems. They argue that the use of AOP for such concerns is hindered first by the need of a high degree of knowledge and expertise in concurrency, and second by the semantic coupling between the modeled objects and the concurrency semantics, *i.e.*, implying a fundamental impossibility to separate concurrency from the modeled objects.

Similarly, Soares, Laureano and Borda [SLB02] showed that the separation of distribution code into aspects using AspectJ over RMI based applications requires a high degree of knowledge of the RMI internals. Furthermore, they show that simple conceptual solutions, like wrapping and redirecting calls to remote objects cannot easily be expressed using sequential aspect oriented languages. For example, consider a system with a single class serving the requests of multiple clients. Such a system may be distributed by separating this class from the middleware handling client requests (this is a typical architecture of web applications implementing enterprise information system). An implementation seems quite simple to be done using ASpectJ and RMI:

```
public aspect RemoteRedirectionAspect {

  pointcut serverFacadeCalls(FacadeServer fs):
        call(* FacadeServer.*(..)) &&
        !call(static * FacadeServer.*(..)) && target(fs);

  Object around((FacadeServer fs): serverFacadeCalls(){
        return proceed(remoteFS);
    }
}
```

The aspect defines a pointcut `serverFacadeCalls` that match all the calls to non-static methods in `FacadeServer` objects. These calls are then wrapped and redirected to a remote reference of the `FacadeServer` objects (`remoteFS`). However this aspect won't work because, due to RMI restrictions, the type of remote objects like `remoteFS` must be a remote interface and not the original `FacadeServer` type. To solve this problem, a programmer could make `FacadeServer` implement a remote interface, changing the original code and add some RMI code to the application. Another solution consists in a wrapping advice for each server method and calling the method explicitly in the remote object `remoteFS`. However, in this case maintenance and extension becomes tedious and error prone (programmers will need to implement an advice for each method on the server). Authors, selected the second approach for their experiments. Similar problems apply to applications with more complex distributed requirements [CC04].

Colyer and Clement have studied the applicability of AspectJ over large-scale middleware [CC04] with heterogeneous distributed requirements. The authors presented an experiment to refactor specific requirements into separate modules from a middleware product line. They attempted to separate the EJB support from application server implementations. Thus, the main idea was to produce application servers, *e.g.*, web application servers, with and without EJB support by pushing a switch in the software product line. The authors have found simplifications in the application structure, and improvements in performance

and memory footprint in the generated applications. However, these results do not include metrics over the modularization structure of the refactored code. It is unclear in particular, if the refactored code was suffering of crosscutting code due to transactions and distribution. Furthermore, the authors do not report on the presence of implicit communication patterns that lead to crosscutting code in the generated code (*e.g.*, see chapter 5).

Finally, several other approaches have adopted sequential AOP on top of distributed frameworks, *e.g.*, JBoss AOP [JBo08a] and Spring AOP [spr08]. A common problem of these frameworks is that the binding between pointcuts and advice is made at configuration time in XML files. These files tend to be verbose, and complex to debug and maintain. Additionally, none of these frameworks present pointcut expressions sensible to localization of events as advocated in this thesis.

2.2.4 Classification and discussion

In this section we have analyzed several non sequential AOP systemas and how they are used to implement distributed applications. We now present the classification of four of these approaches with respect to the taxonomy presented in section 2.1. Table 2.7 shows the classification of the systems: AspectJ using RMI for distribution, Stateful aspects using RMI for distribution, TraceMatches using RMI for distribution, and JBoss AOP (*i.e.*, AOP over a J2EE framework).

These approaches are all based on the remote method invocation communication mechanism, inherited to RMI and J2EE–compliant frameworks. Similarly and even though, RMI and J2EE frameworks support by–reference argument passing, we have classified the passing mechanism as *by–copy*: this is the default behavior, and by–reference behavior requires programmers to modify invasively their code, concretely, by implementing special remote interfaces explicitly in the code. Synchronization mechanism are reused also from the base language. In this case the approaches deal with Java's monitors and the implicit synchronous behavior of remote method invocation.

Regarding the pointcut model, the four aspect systems propose sequential pointcuts that predicate over local join points. Such pointcut models are differentiated only by the level of expressivity. Most notably, atomic vs. stateful aspects (EAOP and stateful aspects) that support the definition of regular expressions over events in the execution of the application.

None of these aspect systems has a remote advice model. All the advice are locals and distributed behavior can only be achieved using the underlying distribution framework.

Most of these systems include several mechanism for aspect instantiation. Having only the number of supported "per" clauses as differentiator. However, JBoss AOP allows programmer to define instantiation factories that can be defined by programmers, thus we have included in the table such value as imperative. Additionally, the models define the deployment mechanism ad a local deployment scope, only systems supporting J2EE are considered to have distributed deployment.

Finally, regarding composition, only local composition mechanisms are provided. The table indicates the different mechanism for each approach. Most notably, stateful aspects use operators to define compositions of different aspects over a particular join point, while AspectJ + RMI and JBoss + J2EE use a precedence mechanism. Tracematches does not include a composition mechanism. The expressiveness of the composition mechanisms is considered as similar to those generated by star-free languages, except for the EAOP approaches.

Taxonomy elements	AspectJ + RMI	EAOP + RMI	TraceMatch + RMI	AOP + J2EE
Communication model				
Communication mechanisms				
Remote method call	yes	yes	yes	yes
Join point propagation	no	no	no	no
Controlled remote advice invocation	no	no	no	no
Group communication	no	no	no	no
Parameter passing modes	by copy	by copy	by copy	by copy
Synchronization model				
Synchronization mechanisms	monitors	monitors	monitors	monitors
Communication timing	synchronous	synchronous	synchronous	synchronous
Asynchronous hypothesis	no	no	no	no
Causal predicates	no	no	no	no
Pointcut model				
Expressiveness:				
Atomic	yes	yes	yes	yes
Sequential control flow	yes	yes	yes	yes
Distributed control flow	no	no	no	no
History based	no	yes	yes	no
Finite-state	no	yes	yes	no
Remote join point support				
All pointcuts	no	no	no	no
Location	no	no	no	no
Paradigm:				
Object-oriented	yes	yes	yes	yes
Remote advice model				
Filtering and ordering	no	no	no	no
Location	no	no	no	no
Synchronization mode	Synchronous	synchronous	synchronous	synchronous
Parameter passing modes	n/a	n/a	n/a	n/a
Proceed support	no	no	no	no
Remote host only	no	no	no	no
Remote and originating host	no	no	no	no
Synchronization mechanisms	Blocking	Blocking	Blocking	Blocking
Mobility	n/a	n/a	n/a	n/a
Reflective access to program state	yes	yes	yes	yes
Aspect model				
Instantiation	Declarative	Declarative	Declarative	Imperative
Declarative				
Per thread	yes	no	yes	yes
Per class	yes	no	no	yes
Per object	yes	no	no	yes
singleton	yes	yes	yes	yes
Per cflow	yes	no	no	no
Per binding	no	no	yes	no
State sharing	no	no	no	no
Deployment	static	static	static	dynamic
Deployment scope	local	local	local	distributed
Weaving mechanisms	static	static	static	load–time
Aspect composition				
Mechanisms	Precedence	Operator	undefined	precedence
Object	aspect	aspect	aspect	aspect
Scope	All	stateful	All	All
Expressiveness	Star-Free	Finite-state	Star-Free	Star-Free
Distributed composition	no	no	no	no

Table 2.7: Classification of sequential aspect-based systems for the implementation of distributed systems

2.3 Aspect oriented programming for distributed applications

We now turn to AOP systems that include explicit mechanisms for distribution at the aspect language level. First, we discuss domain-specific languages for the separation of synchronization and distribution concerns. In particular, we analyze the D framework of languages proposed by Lopes [VL97] (Java, RIDL, and COOL). We then present aspect oriented frameworks that provide AO mechanisms for distribution (JAC [PSD+04] , ReflexD [TT06]). Finally, we discuss a set of language based approaches: DyMAC [LJ06] that extends J2EE with AOP for distribution, remote pointcuts [NST04], and recent approaches dealing with imperative deployment and scoping of distributed aspects.

2.3.1 Domain specific languages

Lopes has proposed D [VL97], a framework of three languages to modularize concurrency and distribution issues in distributed applications. This AO framework used Java as the base language and two additional domain specific languages: COOL and RIDL. Using COOL programmers define syntactical units called coordinators to handle concurrency (trough mutual exclusion specifications). RIDL provides syntactical units called *portals* to address problems of remote invocation and data passing between execution spaces. These two languages provide an example of how, using separate languages for specific concerns, we can modularize crosscutting concerns. Note that this framework weaves three programs (a Java program, a COOL program, and a RIDL program) into one executable. We now present the main characteristics of each of the two domain-specific aspect languages.

COOL

COOL is a language that deals with mutual exclusion, synchronization state, guarded suspension, and notification. Using COOL, programmers write coordinators that are associated to instances of the classes they coordinate. A coordinator is defined as follows:

Coordinator ::=
[`per_class`] `coordinator` *ClassList*
CoordinatorBody

If the optional `per class` declaration is omitted coordinators are associated to one specific object by default. Such coordination is referred to as "coordination per object" and implies that there will be a coordinator instance for each object of the classes in the class list (non-terminal *ClassList*). On the other hand, if `per_class` is used, one coordinator is associated to all the objects of the classes in `ClassList`. The coordinator's (non-terminal *CoordinatorBody*) body may include condition variables, regular variables, self exclusive declarations on methods, declarations on mutually exclusive methods, and method managers (used to define guarded suspension and notifications of thread,*i.e.*, used to define method coordination).

Figure 2.3 shows an example of a coordinator for the `ConnectionPool` class, a typical component of information systems in distributed setting (*e.g.*, a data base connection pool). The coordinator uses the default instantiation (per object). The first line in the body of the coordinator declares the methods `getConnection` and `releaseConnection`, from `ConnectionPool`, as self exclusive, *i.e.*, in the same object neither `putConnection` nor `takeConnection` can be executed by more than one thread at a time (*e.g.*, two threads cannot execute the `getConnection` method concurrently). Line 3 defines `getConnection`

```
1  coordinator ConnectionPool {
2    selfex getConnection, releaseConnection;
3    mutex {getConnection, releaseConnection};
4    condition empty = false, full = false;
5    getConnection: requires !empty;
6         on_exit {
7           if (full) full = false;
8           if (usedConnections == capacity) empty = true;
9         }
10   releaseConnection: requires !full;
11        on_exit {
12          if (empty) empty = false;
13          if (usedConnections == 0) full = true;
14        }
15 }
```

Figure 2.3: COOL coordinator example

Portal	::=	`portal` *ClassName* '{'*PortalBody*'}'
PortalBody	::=	{*RemoteMethodsDecl*} [`default`: TransferableTypeList]
RemoteMethodsDecl	::=	*ReturnType MethodName*(*ParameterList*) ['{' *ObjectTranferDecl* '}']
ReturnType	::=	*JavaType* | `void`
ObjectTransferDecl	::=	*ObjectId*: *Mode*
ObjectId	::=	*Id* | `return`
Mode	::=	`gref` | `copy` ['{'{*CopyDirective*}'}']
CopyDirective	::=	*ClassName SelectionPrimitive VariableList*
SelectionPrimitive	::=	`only` | `bypass`
VariableList	::=	`VariableName` | `all.`*TypeName*

Figure 2.4: RIDL language (excerpts)

and `releaseConnection` as mutually exclusive, *i.e.*, both methods cannot be executed concurrently in one object. In lines 5 to 9 the coordinator defines a method manager for the `getConnection` method. This method manager requires the pool to have at least one connection (`!empty`). Additionally, on the method exit the manager updates the condition variables depending of the values found in variables `usedConnections` and `capacity` from `ConnectionPool`. Note that `full` and `empty` are defined as condition variables in line 4, they can thus be used for guarded suspension purposes. Finally, the coordinator defines a method manager for method `releaseConnection` in lines 10 to 14.

RIDL

RIDL is a domains specific aspect language for remote interfaces used to encapsulate code related to remote invocation and data transfer between different execution spaces. The grammar shown in figure 2.4 shows the essentials of the RIDL language. A RIDL program is composed of a set of *portals*. Portals are syntactic units defining remote methods, parameter passing,

```
1  portal BankSystem {
2    boolean fundTransfer(Account originAccount, Account account, Value value) {
3       //Only strings are copied.
4       originAccount: copy {Account only all.String;}
5       account: copy {Account only all.String;}
6       value: copy {Value;}
7    };
8    Value withdraw(Account account, Value value){
9       //for return object, exclude this edge; this excludes the copies
10      // and breaks nasty cycle.
11      return: copy
12      account: copy {Account only id}
13      value: copy
14   };
15   Value credit(Account account, Value value) {
16      //for return object copy the return value
17      return: copy
18      // for Account, bypass the amount
19      account: copy {Account bypass amount;}
20   };
21 }
```

Figure 2.5: RIDL portal example

and distributed behavior of objects. A portal is declared using the terminal `portal`, an existing java class name, and a portal body. A unique portal instance is associated to each instance of the defining class. The portal body may contain declarations of remote methods, and a default rule for parameter passing in remote calls (see non-terminal *RemoteMethodsDecl*). Such remote method declarations define a subset of the publicly available methods of the defining class. The methods declared in the body will represent the remote interface of the respective object. Besides the method signature (*i.e.*, return type, method name, and parameters), the remote declaration may be extended with a set of explicit rules defining parameter passing modes for the remote method declaration. Those rules can predicate over method return values and parameters. Concretely, programmers may decide that objects are to be passed by reference (see terminal `gref`) or by copy (terminal `copy`). Additionally, the `copy` directive allows programmers to control how the object graph with the respective object as root is serialized and transfered. In particular, programmers can state which fields in the respective object are copied, using the `only` directive, or which fields should not be copied, using `bypass`.

Figure 2.5 shows an example of a RIDL portal for a bank system. The example header defines the portal for objects of type `BankSystem` (see line 1). The body declares as remote three different methods from `BankSystem` class: `fundTransfer`, `withdraw`, and `credit`. Each declaration is modified by data passing directives. For example, in the first remote method declaration the statement in line 4 states that only the fields of type string from objects of type `Account` will be included, when copying the parameter `originAccount`. Similarly, the definition as a remote method of `withdraw` (lines 8 to 13) states that the copies of objects of type `Account` only contain a copy of the `id` field (see line 12) in the copied object graph having as root parameter `account`. Finally, in the declaration of method `credit` the field `amount` from classes `Account` is bypassed, *i.e.*, not serialized (see line 20) when the `account`

value is copied.

Classification

Table 2.8 shows the classification of the D framework according to the taxonomy presented in section 2.1. From the table one of the differentiating features of this approach is the object graph query language that is used to have a fine grained access to the passing by copy behavior. Additionally the language proposes remote method calls as distribution mechanism, mutual exclusion for concurrency, and synchronous communication between processes. The pointcut language is restricted to atomic methods executions. Finally, the framework proposes static deployment, per class and per object instantiation, static weaving, and limited means for composition.

2.3.2 Frameworks for distributed AOP

Many aspect-based approaches have been implemented as frameworks. Frameworks, in general are provided as APIs over an existing programming language. In contrast to programming languages, the use of frameworks typically provide less support for compile time error checking, are more verbose and more difficult to understand. They do, however, not require users to learn a new programming language and are often simpler to extend by new functionality.

Several frameworks providing AOP facilities for distributed applications have been proposed. In this section we present frameworks of two different kinds: first, JAC [PSD+04] and ReflexD [TT06], which provide language extensions on top of Meta Object Protocols (MOPs) and reflective properties of languages; second, DyMAC [LJ06], which extends a distributed component by AOP features.

JAC

Pawlak et al. [PSD+04] introduced JAC (Java Aspect Components) as a framework for aspect oriented programming of distributed applications. JAC extends Java's MOP to provide mechanisms to create aspects, pointcuts, and wrappers to implement crosscutting concerns. Additionally, JAC provides a composition mechanism to allow programmers to configure predefined aspects into existing applications. The main constituents of the JAC framework are *Aspect Components*. Aspect components are hosted in JAC containers that are remotely accessible and define modifications applied to a set of classes in the distributed base application.

Pointcuts are defined as part of aspect components through pointcut methods (aspect components extend framework classes and methods named as `pointcut` are inherited methods). The following code excerpt shows a pointcut definition:

```
pointcut("ALL","Cache","put:void || get:void",
         "MyWrapper",null,true);
```

The pointcut matches the methods `put` and `get` (third parameter), in classes of type `Cache` (second parameter), in all objects (first ALL) independently of their name (in JAC everything may be named, *e.g.*, objects or hosts). Additionally it defines a wrapper class `MyWrapper` that contains the wrapping method `invoke` that defines the behavior, *i.e.*, advice. The last parameter, the boolean value, defines the instantiation method of the wrapping objects. The default of this parameter is `false`, and defines a single wrapper instance (singleton), a value `true` states that a wrapper instance is associated to each matched method. Note that JAC

Taxonomy elements	D framework
Communication model	
Communication mechanisms	
Remote method call	yes
Join point propagation	no
Controlled remote advice invocation	no
Group communication	no
Parameter passing modes	by reference, by copy
Synchronization model	
Synchronization mechanisms	mutual-exclusion
Communication timing	synchronous
Asynchronous hypothesis	no
Causal predicates	no
Remote pointcut model	
Expressiveness:	
Atomic	yes, only method execution
Sequential control flow	no
Distributed control flow	no
History based	no
Remote join point support	no
Paradigm:	
Object-oriented	no
Functional	no
Logic	no
Remote advice model	
Filtering and ordering	no
Location	no
Synchronization mode	Synchronous
Parameter passing modes	by reference and by copy
Parameter passing mechanism	tag language and object graph query language
Proceed support	no
Synchronization mechanisms	Blocking
Mobility	no
Reflective access to program state	yes
Aspect model	
Instantiation	Declarative
Declarative	
Per thread	no
Per class	yes
Per object	yes
singleton	no
Per cflow	no
Per binding	no
State sharing	no
Deployment	static
Deployment scope	local
Weaving mechanisms	static
Aspect composition	
Mechanisms	undefined
Distributed guarantee	no

Table 2.8: Classification of D as an aspect-based systems for distribution

allows to create complex expressions using logical operators and regular expressions. Finally,

the `null` parameter corresponds to a non-specified exception handler.)[4]

In JAC, aspect components are distributed in order to define distribution behavior. Thus, pointcut definitions may contain regular expressions over the host names in an additional parameter (hosts are named at application startup time). For example, an extended version of the previous example could be:

```
pointcut("cache#0","Cache","put:void || get:void",
         "MyWrapper", "host0" ,null,true);
```

In this case the additional parameter with value `"host0"` is a regular expression over the names of hosts. The pointcut is only applied if the aspect is in host with name *host0*. Hence, aspects are not aware of actions taken in remote hosts.

Additionally, distribution is achieved using remote method calls. Figure 2.6 shows an example of an aspect component for replication. The component is declared as a Java class extending the `AspectComponent` class. In the constructor (line 3 to 9) of this class the pointcut is defined on line 7. This pointcut matches the calls to the methods `put` and `get` in objects of type `Cache` and with name `cache#0` (a replica of this object with the same name on each host is created at deployment time).

Then, an inner class `ReplicationWrapper` is defined (lines 12 to 40). This class extends the `Wrapper` class and implements the replication behavior. The class declares the method `invoke` (lines 16 to 18) that is called by default if a join point is matched, this method redirects the call to the method `replicate` (lines 20 to 39). The `replicate` method gets remote references of the replicas of the wrapped object (*i.e.*, object with name `cache#0` on each host) into the vector `replicas` (line 22). Then, in a `for` loop, remote method invocations are performed using reflective access to information (lines 31 to 35). Finally, the original join point is invoked on line 38. Note that using the `proceed` at the end of the method `replicate` simulates an advice defined as a before advice.

This framework includes support for distributed deployment and management of components. However, the activation behavior of advice restricts its expressive power and makes the remote invocations as verbose as those of systems without explicit distributed support. Note that AWED provides a similar mechanism for advice activation, based on host groups, by means of the `on` pointcut. However, it also provides pointcuts aware of remote events, thus augmenting the expressive power and simplifying implementation of remote invocation.

ReflexD

Reflex is a library for structural and behavioral reflection in Java [TNCC03] that has evolved into a kernel for multi-language AOP [TN05]. ReflexD [TT06] is an extension of this framework to support the implementation of distributed AOP approaches.

The Reflex model for behavioral reflection. *Links*, *hook sets*, and *meta objects* are the main elements in the model of behavioral reflection in Reflex. Links bind a set of join points (a hook set) to a meta objects. A hook set is defined as a condition over reifications of language elements, *e.g.*, classes, fields, methods, method invocation, hosts, or constructors. The following statement shows a pointcut example in Reflex:

```
Hookset pcutCallsOnServer = new Hookset(MsgSend.class, new NameCS("Server"),
                                        new NameOs("invoke"));
```

[4] API definitions were taken from JAC's online documentation [Con08]

```
1  public class ReplicationAspect extends AspectComponent {
2
3    ReplicationAspect(){
4     // This aspect is applied on ALL hosts using the
5     // regular expression .*
6
7     pointcut("cache#0","Cache","put:void || get:void",
8             new ReplicationLoadBalancingWrapper(this, replicaExpr), ".*" ,null,true);
9    }
10
11   // Inner class defining the advice
12   class ReplicationWrapper extends Wrapper{
13     Vector replicas = null;
14     boolean retry = true;
15
16     public Object invoke(MethodInvocation invocation){
17       return replicate((Interaction) invocation);
18     }
19
20     public Object replicate(Interaction interaction) {
21       if (doFill) {
22       replicas = Topology.getPartialTopology(".*").getReplicas(
23                       interaction.wrappee);
24         retry = false;
25       }
26       if (replicas.size() == 0) {
27         // none replicas where found, we perform a local call and
28         // will try to get them again on the next call
29         retry = true;
30       }
31       for(int i=0; i < replicas.size(); i++) {
32           ((RemoteRef) replicas.get(count++)).invoke(
33                interaction.method.getName(),
34                interaction.args);
35       }
36
37       // we perform always the local call
38     return proceed(interaction);
39      }
40   }
41 }
```

Figure 2.6: Aspect component for replication in JAC framework

The example declares a new hook set that will match *message send* actions involving the `invoke` method from class `Server` as target (see the name-based class and operation selectors: `NameCS` and `NameOs`). The model provides a hierarchical representation of source code at the byte-code level. Thus, a `RPool` objects serves as root of an object graph which has several `RClass` objects representing classes which have access to their respective members,*e.g.*, fields, methods, constructors. All the elements provided as reified objects (*e.g.*, `RField`, `Rmethod`) may be used to define hook sets.

As mentioned before Reflex provides *Links* to bind hook sets to specific actions defined in objects. A link for the previous hook set may be:

```
Link trace = Links.get(pcutCallsOnServer, new Tracer());
trace.setControl(Control.BEFORE);
trace.setCall("Tracer", "log", Parameter.THIS);
```

This example creates a `trace` link to bind the `pcutCallsOnServer` hook set to a new `Tracer` meta object. Then the code sets the meta object control to `BEFORE` in order to execute the advice before the corresponding matched execution point. Finally, the code explicitly defines the method `Tracer.log` to be used as advice and the predefined parameter `THIS` is passed.

ReflexD. To address distribution issues, ReflexD provides extended features for hook sets, links, and actions. In particular, a new element reifiying the executing process has been added to the reflective model. Concretely, Reflex-enabled VMs (ReflexD processes) have been reified into `RHost` objects. These objects allow programmers to define selectors that can then be used in hook sets. A simple example of a selector is defined as follows:

```
public class DataBaseServerSelector implements HostSelector{
  public boolean accept(RHost aHost){
    return "true".equals(aHost.getProperties().get("isDataServer"));
  }
}
```

This selectors implements the `HostSelector` interface provided by ReflexD. Then, it evaluates the property whether the given process is or is not a data base server. This constructor can then be used as part of a hook set as follows:

```
Hookset pcutCallsOnDataServer = new Hookset(MsgSend.class, new NameCS("Server"),
                                    new NameOs("invoke"), new DatabaseServerSelector());
```

The hook set `pcutCallsOnDataServer` matches all calls to method `invoke` on objects of type `Server` on data base hosts.

Once a hook set is defined to match events in different hosts, a link can be defined to bind an action to these events. The following code excerpt defines a link that binds the previous host set to a remote meta object:

```
RHost host = RHosts.get("178.1.3.4:4523","ReplicationServer");
Link remoteTrace = Links.get(pcutCallsOnDataServer, new MODefinition.Class("Replication",
                                    new ExecHost(host)));
remoteTrace.setCall("Replication", "replicate", new ByRef(Parameter.THIS)),
                        Parameter.HOST);
```

In this example, a remote `Replication` object is created on the replication server and bound to the hook set `pcutCallsOnDataServer`. ReflexD follows Java's Remote Method Invocation (RMI) semantics for parameter passing, that is, only remote objects are passed by reference.

```
1  ao_composition {
2  AdvisingComponent : ReplicationComponent;
3  Scope : Singleton;
4  Binding {
5   Pointcut {
6    Kind : execution ;
7    MethodMessage : * * ( . . ) ;
8    Caller {
9      Hostgroup : clients; }
10   Callee {
11     Interface : IERPService ;}}
12  Advice {
13   Kind : around ;
14   MethodMessage : ReplicateCallIntoReplicationService;
15  }}
16 }
```

Figure 2.7: Replication in DyMAC

The example also shows how ReflexD allows to set the advice class and the advice method remotely and transparently, and how parameters are passed to the framework.

Additionally, ReflexD provides mechanisms to deal with scope, and instantiation of distributed aspects. ReflexD offers the instantiation of one aspect instance per object, per class, and per host. Finally, ReflexD provides means to explicitly and implicitly instantiate the action objects, including remote deployment.

DyMAC

Lagaisse and Joosen proposed DyMAC [LJ06] an aspect oriented middleware platform with an aspect-component model for distributed applications. This framework address the problem of software development by means of composition of third parties components (à la J2EE or .NET), and the composition of crosscutting services by means of AO mechanisms. One of the main feature of Dymac is that it provides aspect based abstractions (*e.g.*, pointcuts, advice) to deal with the elements of the component model. Thus, in contrast to, *e.g.*, JBoss AOP, aspects are not applied to the underlying class model, but instead to the component model. To this purpose, DyMAC provides pointcuts to predicate over calls and executions of remote method invocations, and advice defined in remote components.

Concretely, the model provides components to define advice behavior, and application descriptors for aspect oriented compositions. These application descriptors are composed of a set of bindings where each binding defines a pointcut, a component and an advising method, and the kind of advice (before, after, around). Figure 2.7 shows how a replication service in an Enterprise Resource Management (ERP) application can be implemented using DyMAC. In line 3, the component is instantiated as a singleton (*i.e.*, one instance in the distributed system). Then the composition defines a pointcut that matches all method calls (lines 6 and 7), of all hosts from the *clients* host group that implement the *IERPService* interface. Finally, the composition defines an `around` advice using the `ReplicateCallIntoReplicationService` method from the `ReplicationComponent` .

The previous example shows already the main features of DyMAC. First, DyMAC proposes several explicit scoping modes: one instance in the system (`Singleton`); one instance

per host or host group, per application domain, per application, per component, and one instance per logical (distributed) thread. Regarding the pointcut language, the DyMAC model proposes elements to match method calls, and method executions. Additionally, pointcuts allow to predicate over component context features, *e.g.*, component names, or caller and callee names. Similarly, pointcuts may evaluate infrastructure context information, *e.g.*, host names, or host groups. Regarding advice definitions, methods defined as part of components are used. Such methods and components are bound to specific pointcuts stating a particular kind of advice (around/before/after). Finally, the model proposes the inclusion of advising restrictions that annotate the advice-methods (not shown in the example). Such annotations are part of the component interface and are controlled automatically by the framework. In particular, a component method may include the types of advice that it supports, requires, or prohibits. Finally, the model provides remote `proceed` semantics, thus a `proceed` invocation in an `around` advice will execute the method call or execution corresponding to the original joinpoint.

2.3.3 Classification ad discussion

Table 2.9 shows the classification of the three frameworks for distributed AOP that we have analyzed: JAC, ReflexD, DyMAC. One of the fundamental differences with AWED is that join point matching in these approaches has local semantics (see item `Remote join point support`). Thus join points are matched in local machines, even though advice may be triggered on a remote host. For example, in JAC aspect and wrapper instances are replicated on each host, and thus the distributed behavior is implemented based on this model: each replica matches the join points in its respective host. In ReflexD, links bind pointcuts to remote meta objects. This means that either the link is defined in the corresponding host or it has to be defined in a link repository, hence augmenting the complexity of the deployed architecture. DyMAC, on the other hand, matches remote calls in the component framework, thus distribution of the underlying application is assumed, and is complemented with the possibility of using remote components in the aspect compositions.

2.4 Language support for distributed aspects

Currently, there are few approaches that provide language support for distributed aspects. In this section we discuss two notable exceptions: support for remote pointcuts and for the instantiation as well as scoping of distributed aspects.

2.4.1 Remote pointcuts in DJCutter

In 2004, Nishizawa, Chiba, and Tatsubori [NST04] presented the DJCutter language, introducing the concept of "remote pointcuts" to designate pointcut constructs that could match join points that occur remotely in a different application space.

DJCutter is an AspectJ-like compiler and runtime supporting explicit distribution. This approach was the first to consider a model of remote join points where any join point, independent from their physical location, can be matched by a pointcut definition. (However, the DJCutter model was rather limited in that advice was always executed on a specific central host; this property was motivated by its intended application domain: remote testing). DJCutter also provided aspect weaving at load time, and remote deployment of aspects.

Taxonomy elements	JAC	ReflexD	DyMAC
Communication model			
Communication mechanisms			
Remote method call	yes	yes	yes
Join point propagation	no		
Controlled remote advice invocation	no	no	no
Group communication	no	no	yes
Parameter passing modes	by ref., by copy	by ref., by copy	by ref., by copy
Synchronization model			
Synchronization mechanisms	mutual-exclusion	mutual-exclusion	mutual-exclusion
Communication timing	synchronous	synchronous	synchronous
Asynchronous hypothesis	no	no	no
Causal predicates	no	no	no
Remote pointcut model			
Expressiveness:			
Atomic	yes	yes	yes
Sequential control flow	no	yes	no
Distributed control flow	no	yes	no
History based	no	no	no
Remote join point support	no	no	no
Paradigm:			
Object-oriented	yes	yes	no
Component based	yes	no	yes
Functional	no		
Logic	no		
Remote advice model			
Filtering and ordering	no	no	no
Location	yes	yes	yes
Synchronization mode	Synchronous	Synchronous	Synchronous
Parameter passing modes	by ref., by copy	by ref., by copy	by ref., by copy
Parameter passing mechanism	Framework default	Framework defined	Framework default
Proceed support	yes	yes	yes
Synchronization mechanisms	Blocking	Blockin	Blocking
Mobility	no	no	no
Reflective access to program state	yes	yes	yes
Aspect model			
Instantiation	Imperative	Imperative	Declarative
Mechanisms			
Per thread	no	no	yes
Per class	no	yes	yes
Per object	no	yes	yes
Per join point	yes	no	no
singleton	yes	yes	yes
Per cflow	no	no	no
Per binding	no	no	no
State sharing	no	no	no
Deployment	dynamic	dynamic	dynamic
Deployment scope	global	remote hosts	group of hosts
Weaving mechanisms	load–time	load–time	framework–based
Aspect composition			
Mechanisms	Precedence	undefined	undefined
Object	aspect	n/a	n/a
Scope	All	n/a	n/a
Distributed guarantee	no	no	no

Table 2.9: Classification of AOP frameworks for distribution: JAC, ReflexD, and DyMAC

An example of a DJCutter pointcut for monitoring accounts withdrawals is the following:

pointcut withdrawMonitor(): **call**(**void** Account.withdraw(**int**))

Here, a `call` pointcut matches the calls to the method `withdraw` of class `Account`. The pointcut will match such calls in any participating host. Hence, a single pointcut definition written in a non-distributed fashion will serve to match join points in a distributed setting.

The language provides the `hosts` pointcut constructor that allows matching to be restricted to sets of hosts. Finally, DJCutter also introduced a notion of distributed `cflow` that was aware of the distributed deployment. The implementation of this pointcut uses RMI customized socket features to pass information of the call stack to the aspect server.

Taxonomy elements	DJCutter
Communication model	
Communication mechanisms	
Remote method call	yes
Join point propagation	yes
Controlled remote advice invocation	no
Group communication	no
Parameter passing modes	by reference, by copy
Synchronization model	
Synchronization mechanisms	mutual-exclusion
Communication timing	synchronous
Asynchronous hypothesis	no
Causal predicates	no
Remote pointcut model	
Expressiveness:	
Atomic	yes
Sequential control flow	yes
Distributed control flow	yes
History based	no
Remote join point support	yes
All pointcuts	yes
Location	Single host, all hosts
Paradigm:	
Object-oriented	yes
Functional	no
Logic	no
Remote advice model	
Filtering and ordering	no
Location	no
Synchronization mode	Synchronous
Parameter passing modes	by copy, and by reference
Parameter passing mechanism	only `Remote` objecs are passed by reference
Proceed support	no
Synchronization mechanisms	Blocking
Mobility	no
Reflective access to program state	yes
Aspect model	
Instantiation	Declarative
Mechanisms	
singleton	yes
State sharing	no
Deployment	dynamic
Deployment scope	global, local
Weaving mechanisms	load–time
Aspect composition	
Mechanisms	undefined
Distributed guarantee	no

Table 2.10: Classification of DJCutter as an aspect-based language for distribution

Table 2.10 shows the classification of DJCutter according to our taxonomy of distributed

aspects.

This thesis, especially the AWED model and system, has been motivated initially by the work on DJCutter. It extends its predecessor approach in several ways. First, AWED extends the pointcut model by providing a richer model of remote pointcuts, a model for fully distributed cflow, and pointcuts for distributed trace matching using state machines. Second, we provide a model for distributed advice including the manipulation of host groups. Third, AWED provides a decentralized model for advice execution, where aspects are distributed when needed. Finally, AWED provides a model allows restrictions to be placed on non-deterministic executions in distributed systems using features for message ordering and expression of causality between messages.

2.4.2 Aspect scoping and instantiation

Finally, our work has also strived for new mechanisms for scoping and instantiation of distributed aspects.

The scope of an aspect defines the set of joinpoints that may be matched by an aspect (see, *e.g.*, [AGMO06, ET08, FB07, ETFD+08]). Even though an aspect scope may be restricted using pointcuts definitions, several aspect languages have proposed explicit scoping mechanisms. For example, in AspectJ [KHH+01], if an aspect is declared using a *pertarget(Pointcut)* clause, an aspect instance is created for each individual target object matched by the pointcut definition. For such aspects, an advice will be executed at join points only occurring in the context of the corresponding target object. Other possible example of *per* clauses attach an aspect instance to a specific control flow or to the *this* object of a specific join point. In these kind of models the concepts of instantiation and aspect scope are mixed.

A different model is proposed by Alan et al. [A+05] as part of the tracematch approach. In their aspect language, a tracematch is used to match regular sequence of events (see section 2.2.2). The language allows free variables to be used in the definition of the regular expressions. Free variables can then be bound to different objects and are a source of parametrization in the regular expression. Using this technique, a single tracematch can match all possible traces occurring in a program, even with different bindings for the free variables. Note that the language does not create an instance for each possible trace, instead it proposes an efficient algorithm to handle all possible bindings.

The two approaches presented before define aspect scopes and instances at build time. To the contrary, CaesarJ [MO03] supports dynamic aspect instantiation and scoping per thread: aspects are instantiated at runtime as defined programmatically using deploy blocks:

deploy(anAspect){ ... }

In the above example, the aspect `anAspect` can potentially be applied to all join points produced in the dynamic extension of the code defined in the statement body. Other approaches like AspectScheme proposes lexical scoping for aspect deployment, thus an aspect will see all the join points produced in the lexical scope of a code block. Similarly, CaesarJ provides explicit languages mechanism to deploy an aspect in a specific thread execution. However, in CaesarJ the scope of an aspect is always either global, allowing aspects to see all events in a program, or limited to a specific thread of control. A *perobject* deployment strategy, for instance, is not provided.

CaesarJ also proposes an approach for distributed deployment of aspects. The approach allows the programmer to send an aspect to a remote process running CaesarJ. The deploy-

```
1    CaesarHost host = new CaesarHost("rmi://awed.org/Server/");
2    ReplicationAspectAgent raa = new ReplicationAspectAgent();
3    raa.setAdvicedObject((Cache)host.resolve("MyCache"));
4    localCache.observe(raa);
5    host.deployAspect(raa);
```

Figure 2.8: Distributed deployment in CaesarJ

ment scope will be the process where the aspect was deployed. Figure 2.8 shows a code example of remote deployment in CaesarJ. Line 1 defines a `CaesarHost` object. Line 2 instantiates an aspect `ReplicationAspectAgent`. Lines 3 initializes an instance of a `Cache` object in the remote host. This object will be the object monitored for application by the aspect. A local cache is set to observe the aspect in line 4. Finally, the aspect is deployed in the remote host. Once deployed, the aspect will match join points in the remote host and will call the method update in the observer `localCache`. CaesarJ transparently manages remote references, however the semantics of pointcut matching is a local one.

All previous approaches present different but limited semantics for aspect scoping. In a recent proposal, Tanter [ET08] generalized these approaches proposing an explicit way to deal with scope of dynamically deployed aspects. Using this approach, the programmer can specify explicitly the scope of an aspect using deployment strategies based on orthogonal dimensions of propagation (*e.g.*, along the call stack, and as part of delayed evaluation) and join point filtering. A programmer can then create aspects that will propagate following the dynamic activity of an object *e.g.*, the control flow, or she can create aspects that will be deployed using the delayed evaluation *e.g.*, affecting the behavior of objects created at some later point in time.

Classification. Table 2.11 shows the classification of CaesarJ according to the taxonomy.

Taxonomy elements	CaesarJ
Communication model	
Communication mechanisms	
Remote method call	yes
Join point propagation	no
Controlled remote advice invocation	no
Group communication	no
Parameter passing modes	by reference, by copy
Synchronization model	
Synchronization mechanisms	mutual-exclusion
Communication timing	synchronous
Asynchronous hypothesis	no
Causal predicates	no
Remote pointcut model	
Expressiveness:	
Atomic	yes
Sequential control flow	yes
Distributed control flow	yes
History based	no
Remote join point support	no
Paradigm:	
Object-oriented	yes
Functional	no
Logic	no
Remote advice model	
Filtering and ordering	no
Location	no
Synchronization mode	Synchronous
Parameter passing modes	by reference and by copy
Parameter passing mechanism	Transparent references of any object
Proceed support	yes local
Synchronization mechanisms	Blocking
Mobility	weak
Reflective access to program state	yes
Aspect model	
Instantiation	Imperative, Declarative
Mechanisms	
Per thread	yes
Per class	no
Per object	yes
singleton	yes
Per cflow	yes
Per binding	no
State sharing	no
Deployment	dynamic
Deployment scope	remote host
Weaving mechanisms	load–time
Aspect composition	
Mechanisms	Program
Object	cclass
Scope	stateful
Distributed guarantee	no

Table 2.11: Classification of CaesarJ as an aspect-based systems for distribution

Chapter 3

Crosscutting and evolution of JBoss Cache

One of the major open questions of AOSD is how stable aspect-oriented designs and programs are in the presence of evolution of the underlying application. This question has been investigated only rudimentarily, for instance, by Coady and Kiczales [CK03] study on evolution of crosscutting concerns in operating systems, or the study on EJB support over application server product lines by Coyler and Clement [CC04].

As part of this PhD work, we have investigated JBoss Cache over three years. While being initially simply an medium-sized real-world application to which our aspect model has been applied, JBoss Cache has undergone two major evolution steps during that period. In this chapter, we present a detailed introduction to the (crosscutting and inter-dependent) functionalities of replication and transaction management in JBoss Cache on an architectural and implementation level. As a second contribution, we provide a detailed study of the relationship between the two evolution steps and the crosscutting functionalities. Summarizing our main result, we show that, although one of the evolution steps mainly introduced additional support for the modularization of transactional behavior, the crosscutting characteristics of replication and transactions as well as significant problems in the understanding, maintenance and evolution of JBoss Cache persist in all versions of the software.

Concretely, our analysis clearly pinpoints three specific problems of JBoss Cache

- We show how fundamental concerns in middleware development, *e.g.*, replication and transactions, that can be handled by a simple conceptual architecture, are heavily scattered and tangled in the JBoss Cache implementation.

- We show that current encapsulation techniques and derived abstractions (*e.g.*, classes, packages, or design patterns) are insufficient to address the encapsulation of these crosscutting concerns.

- We show that the main crosscutting properties of replication and transactional behavior in JBoss Cache have been largely proven to be resilient to evolution, even to OO means geared explicitly towards resolving this modularity issue.

Note that we strongly reckon that these problems are typical for Java-based middlewares. We have, in particular, performed similar studies, albeit of more limited scope, in the context

of other middlewares, *e.g.*, ActiveMQ, the Apache messaging middleware framework, see Chapter 6.

Our study involved an analysis of JBoss Cache over three major versions during which this infrastructure has evolved from 13,000 lines of code (13 KLOC) in the first version we considered to more than 40 KLOC in the third version. In the following, we first introduce the main problems as well as architectural and implementation artifacts of JBoss Cache, version 1.2.1 (release date: 2005-03-14). We then study how these issues and artifacts have evolved via versions 1.4.1 (release date: 2006-11-14) to version 2.0.0 (release date: 2007-08-08).

The chapter is structured as follows. Section 3.1 presents an overview of replication and transaction management in JBoss Cache. In section 3.2 we give a detailed analysis of the main architectural and implementation artifacts of JBoss Cache and show that replication and transactional concerns are heavily crosscutting. Section 3.3 provides evidence that the transformation of JBoss Cache performed over the two evolution steps has not been sufficient to address the problem of modularization of transactional behavior and replication, even though dedicated OO abstractions have been introduced for the management of transactional behavior. Finally, we present a quantitative analysis of the crosscutting characteristics of JBoss Cache.

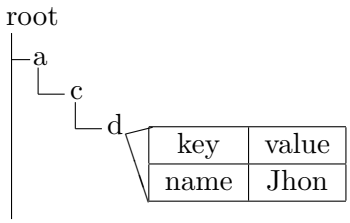
3.1 Overview of replication and transaction management

A replicated cache is a data structure (*e.g.*, a tree, a vector, a hash table, a queue) that has been augmented with functionalities supports replication of data in a distributed system in order to speed up accesses to otherwise remote data and may support additional functionality such as data persistence and transactional behavior. In the case of JBoss Cache, the data structure storing replicated data is a tree with a hash table on each node.[1]

The implemented data structure has a simple write and read protocol. An example of the usage of JBoss Cache Java's API is

cacheInstance.put("\a\c\d", "name", "Jhon")

Here, the object `cacheInstance` receives a call to method `put` with three string objects as parameters. The first parameter represents the path to the node where the information is going to be stored, in this case the information is stored in node `d`, that is reached walking from the root node to node `a`, then to child node `c`, and finally to child node `d`. Once in node `d` the string parameter "name" is used as a key to store the string object "Jhon" in node's hash table. The following graphical representation shows the tree data structure after the execution of the statement:



[1]Note that the tree structure allows a fine-grained mapping of object graphs. Thus replication can be done efficiently over atomic elements of the object graph, *e.g.*, a field in an object.

To obtain a fully distributed middleware data should be also replicated to other caches in the cluster. To achieve this purpose the data structure implementation has been augmented with code for replication and transactions, *i.e.*, the coherence of the data in the caches forming a common cluster is ensured using a transactional model of concurrency control [EGLT76, Gra78]. In its basic behavior JBoss Cache can either be configured to be local, in which case no data is replicated to other caches on other machines, or it can be global, which means that all changes are replicated to all the other caches (on all other machines) that are part of the cluster. Regarding transactions JBoss Cache can be configured to use pessimistic [EGLT76, Gra78] or optimistic locking [KR79]. Pessimistic locking algorithm locks the nodes on the tree that are participating in a transaction (all along the transaction), and commits only if no conflicts are found in remote nodes (this algorithm is more efficient if the probability of two transactions accessing the same node is high). In optimistic locking, conversely, the system creates a copy of each node involved in the transaction, then at commit time the node is locked and the transaction is committed if all involved nodes haven't been changed and if there are no conflicts in remote nodes (this algorithm is more efficient if the probability of two transactions accessing the same node is low). In both cases, once a transaction is finished on the local machine, a two phase commit protocol [LS76] is initiated to replicate the new data. We can modify our previous example to use transactions as follows,

```
tx.begin();
cacheInstance.put("\a\c\d", "name", "Jhon");
tx.commit();,
```

in the example the call to method `put` is now guarded by a `begin-commit` block (`tx` is a transaction object). In this case the `put` method is not replicated and is only applied on the local cache, then, once the `commit` statement is reached the two phase commit protocol is started. First, a prepare statement is sent, with all the information of the transaction, towards all the cache instances in the hosts participating in the replication cluster. If all the caches can acquire the necessary local locks to enable the modifications, a commit message is sent to finalize the transaction, otherwise the transaction is rolled back on all hosts.

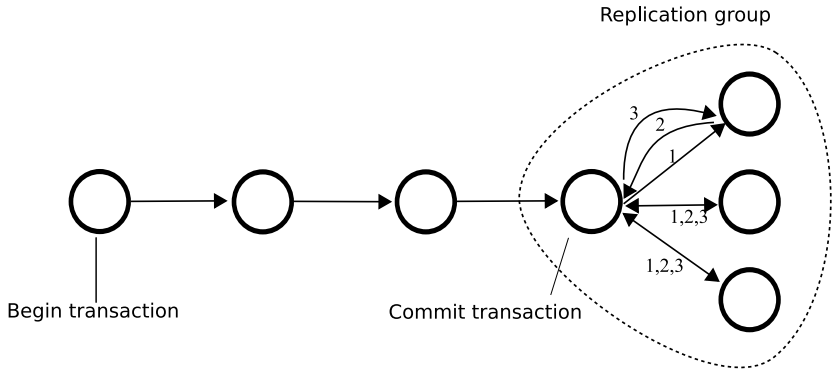


Figure 3.1: Architecture of transaction handling with replication in JBoss Cache

Figure 3.1 presents a high-level pattern-based view of the corresponding runtime architecture of JBoss Cache. In the figure, a transaction is triggered by a specific method call represented by the first node in the pattern. Then successive calls to `get`, `remove` or `put` methods on the cache are executed and the information is stored for further replication. Once the end of a transaction is reached, the originating cache engages a two phase commit

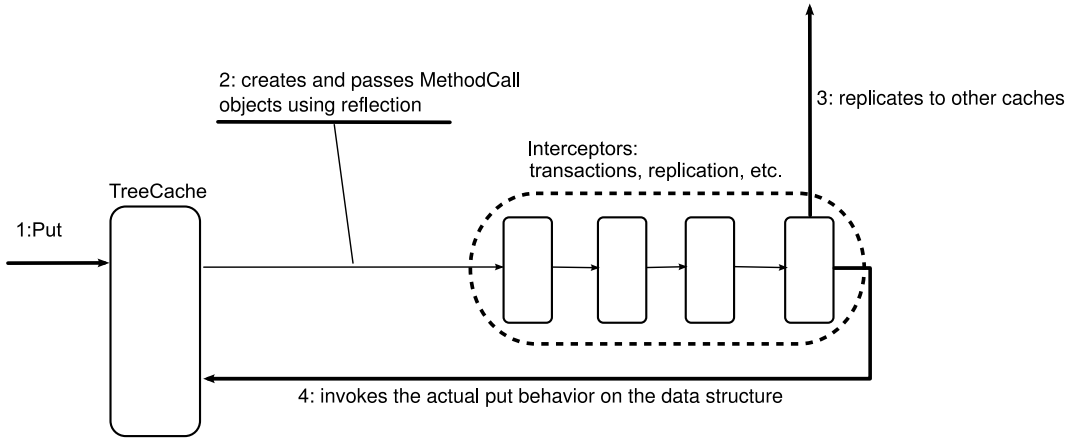


Figure 3.2: Interceptor chain pattern implementation in JBoss Cache.

protocol. In such a protocol the originating cache sends a prepare message with the transaction control information (edges numbered 1 in the right part of the figure), followed by answers from all hosts confirming agreement or non agreement (edges numbered 2). Finally, the originating cache sends a final commit or a rollback message depending on the answers it received (edges numbered 3).

3.2 JBoss Cache implementation: principles and crosscutting

The runtime description above and system architecture of JBoss Cache shown in figure 3.1 provide an easy-to-grasp high-level view of its operation. However, a crucial question is how faithfully such abstract representation can be transposed into an a object-oriented (Java-based) implementation. In this section we answer this question in three steps. We first present the design principles guiding the JBoss Cache implementation. It turns out that the implementation does not rely on standard Java-based structuring mechanisms (such as packages, classes and objects) but uses so-called interceptors, a more complex reflection-based means for application structuring. Second, we provide first evidence for the crosscutting nature of JBoss Cache's main functionalities. Third, we discuss why JBoss AOP, a component of the JBoss application server, is not suitable to address the crosscutting issues of JBoss Cache. Basically, JBoss AOP falls short because it is is a subset of the (sequential) AspectJ model that is not appropriate for the modularization of non-sequential crosscutting functionalities as discussed in chapter 2.

3.2.1 Design principles

JBoss Cache uses an interceptor filter pattern as a main structuring mechanism at the code level, see figure 3.2. The main idea behind this pattern is that method calls to the cache data structure are pre-processed by a chain of filters, where each filter implements a specific concern, *e.g.*, replication or transactions. In the figure, class `TreeCache` implements the tree data structure and the filters are implemented by classes called interceptors. Each method invocation to a `TreeCache` object is then processed by the elements of the interceptor chain. For example, a call to the method `put` (edge numbered 1) on the cache, is first transformed,

```
1 public Object put(Fqn fqn, Object key, Object value)
2    throws CacheException {
3       GlobalTransaction tx=getCurrentTransaction();
4       MethodCall m=new MethodCall(putKeyValMethodLocal,
5                                   new Object[]{tx, fqn, key,
6                                   value, Boolean.TRUE});
7       return invokeMethod(m);
8    }
```

Figure 3.3: Low-level transaction handling in class `TreeCache`

using reflection, into a `MethodCall` object and is then passed to the chain of filters (edge numbered 2). Conceptually, each filter then fully implements its respective functionality, *e.g.*, the replication filter replicates the class to other caches (edge numbered 3). As we show later, however, the modularization of the key functionalities is far from perfect. Finally, once the call has passed all the filters in the filter chain, the actual behavior is invoked in the data structure (edge numbered 4).

3.2.2 Implementation structure and crosscutting issues

The implementation of this design principles is subject to severe crosscutting issues. Such issues are present in the implementation of the basic methods managing the replication data structure but also in the replication and transaction filters themselves.

Tangling in the replication data structure. This code structure is apparent in the low-level cache manipulation methods of class `TreeCache`. For example, figure 3.3 shows the definition of method `put`. As defined in the method signature, it returns an object and receives, as parameters, a fully qualified name, a `key` object, and a `value` object. In the body definition the code has to get the transactional context (Line 3), modify it if necessary, create (using reflection) an object of type `MethodCall` (Lines 4 to 6), and invoke it reflectively, that is, pass it along the interceptors chain.[2] Note that this description already strongly hints at a code tangling problem of the implementation of the `put` method (and similarly, the other low-level cache management methods). In particular, transactions, the creation of an object for replication (an object of type `MethodCall`), and the calls to and from the filter patterns are tangled.

Crosscutting and filters. Figure 3.4 shows two different representations of the code structure of the filter patterns in JBoss Cache 1.2.1 . The class diagram (Figure 3.4a) shows the main classes participating in the implementation: a class representing the main data structure (`TreeCache`), that use a chain of interceptors represented by the class `Interceptor` that can be chained to other `Interceptor` objects (see the aggregation labeled "next" that represents the field `next` of type `Interceptor`). The `Interceptor` class is then specialized by specific interceptor classes (*e.g.*, classes implementing replication, transactions or locking). The class diagram also shows that the class implementing transparent caching of POJOs

[2]This `put` method differs slightly form the one presented in section 3.1: there, the method receives three `String` objects as parameter, however the implementation of that method redirects the call to the one presented here after processing and converting the `String` objects.

(TreeCacheAOP) is a specialization of the main data structure (see an explanation of JBoss AOP and JBoss cache below in 3.2.3).

Even though, the implementation of code shown before follows well known practices for design and modularization *i.e.*, design patterns and inheritance, an analysis of the code shows that modularization is not achieved and the code is subject to crosscutting by the code for replication and transactions, see figure 3.4b. The figure, a standard crosscutting view generating using the AJDT Eclipse environment for AspectJ, depicts the scattering of replication and transaction code in the main class `TreeCache` (represented by the leftmost column in the figure) and in the classes belonging to the interceptor package (right block in the figure). Replication-related code is colored gray, transaction-related code is marked black. The figures clearly exhibit scattering and tangling of replication and transactional code with respect to each other and with respect to the functional code of the cache. Furthermore, the code related to the implementation of the interceptor filters — that was added to help modularization — is subject to severe crosscutting itself. Thus the mechanism used for modularization generates itself modularization problems. Concrete figures that provide evidence for these claims are given in section 3.3.3. In the addition to crosscutting directly due to replication and transactional code, calls between the interceptor package and the remaining code parts , which we have not included here to stress our main point, are also crosscutting.

3.2.3 Caching of POJOs and JBoss AOP.

Besides the filters discussed above, JBoss AOP, another AO-related component of the JBoss application server could potentially have been used to improve the modularization properties of JBoss Cache. JBoss AOP is a framework for Aspect-Oriented programming of JBoss and is used in JBoss Cache to cache "plain old" Java objects ("POJOs") in a transparent manner. Concretely, this mechanism handles object inheritance, aggregation, as well as the object graph, *e.g.*, for serialization, in the context of caching.

However, this use of JBoss AOP does not contribute to the goal of a better modularization of the cache: JBoss AOP is solely used to facilitate the use of JBoss Cache in an application but does not address modularization or extension of JBoss Cache core functionalities. Conceptually, JBoss AOP is not suited to achieve this goal: as a subset of the (sequential) AspectJ model, any implementation involving several cache members, which is at the very heart of the crosscutting problem we want to solve, would be subject to the problems presented in Sec. 1.1.

3.3 Evolution and crosscutting in JBoss Cache

We now analyze how crosscutting concerns evolved through three different versions of JBoss Cache. In particular we study how the design principles and implementation strategies shown before have evolved and what the effects on the crosscutting characteristics of JBoss Cache's core functionality has been. Concretely, we show that introduction of new OO abstractions explicitly aimed at the improvement of modularization properties combined with re-engineering and refactoring techniques have proven insufficient.

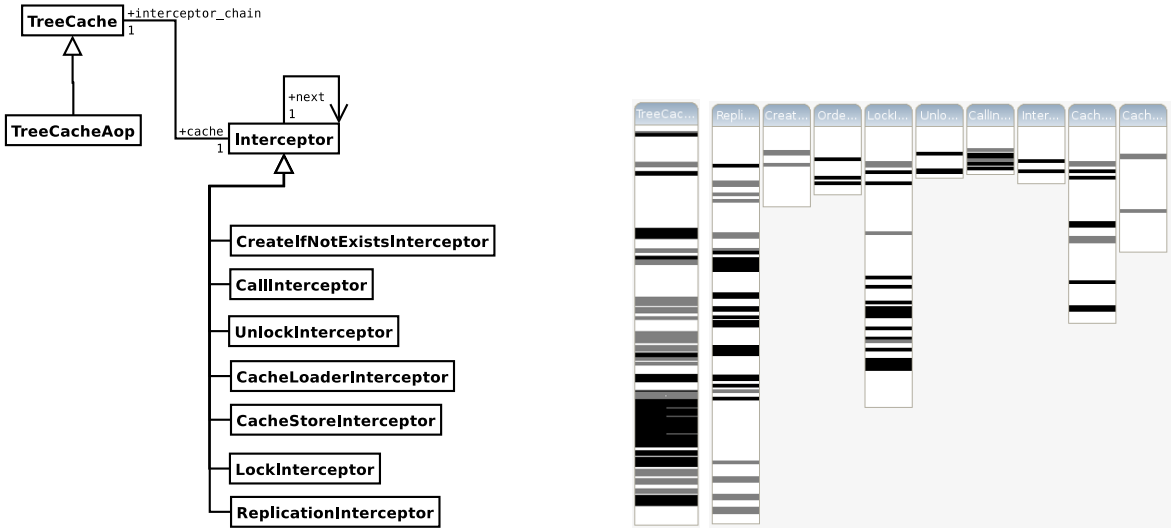


Figure 3.4: Code structure of JBoss Cache: a) Class diagram of filter pattern implementation, b) Crosscutting diagram of scattered and tangled code for distribution and transactions.

3.3.1 Evolution of the interceptor framework

In this section we compare the evolution of main features and code structures (classes) in three different versions of JBoss Cache: version 1.2.1 (that is used in sections 3.1, 3.2, and 3.2.2 to introduce replication and transaction management, its architecture, and the main implementation structures), version 1.4.1, and version 2.0.0. (Henceforth, we refer to the three versions as 1.2, 1.4 and 2, respectively.)

Table 3.1 shows the commonalities and differences between the three versions with respect to the interceptor mechanism that is the main modularization mechanism in JBoss Cache. In the first column, it shows the list of interceptors (*i.e.*, filters) classified according to the main features of JBoss Cache. The table make explicit which interceptors have been present in which version of JBoss Cache. For example, line three indicates that the interceptor `BaseInterceptor` appeared in version 1.4 but then disappears in version 2. Additionally, the table provides information about the inheritance relationships between interceptors: inheritance is indicated using indentation; the class `Interceptor` (line 2), for example, is the base class in the hierarchy (the super class of all other classes) (The class `OrderedSynchronizationHandler`, see line 17, is not a subclass of the `Interceptor` class, however I included it in the list under the hierarchy because it is located in the package `Interceptors`, and is related to transaction management.) Note that the information about sub-classing (inheritance) is given for the latest version (2) of JBoss Cache, some of the classes can be located in different hierarchies in previous versions. This is discussed to some extent later in this section.

Table 3.1 provides useful information regarding evolution, in particular about package and class extensions, and refactorings. Regarding extensions, comparing versions 2 and 1.2 we can see a large increase in functional and non functional features. Version 1.2 included nine classes to implement interceptors, while version 2 comprises 34 such classes. Version 2 includes new support for replication, in particular, support for data gravitation,

a replication mechanism towards "buddy" nodes a subset of the nodes in a cache that have explicitly announced their interest in the data, see line 8. Similarly, line 9 presents the interceptor that adds support for replication by invalidation: instead of replicating a change a change is detected, the respective node is invalidated and this invalidation information (only the data indicating what node is invalidated) is replicated to other caches, forcing them to read information from persistent storage. Other extensions introduce the following: support of optimistic locking for transactions (lines 18-22); support for node eviction (line 31),*e.g.*, based on a timeout policy; support for passivation (line 33), *e.g.*, storing state on disk when memory gets low; and support for service management (line 35).

Table 3.1 also hints at refactorings. For example, classes that have appeared and disappeared are: class `BaseInterceptor` (line 2), class `CreatIfNotExistsInterceptor` (line 5), class `BaseCacheLoaderInterceptor` (line 28). An example regarding renaming is found in class `LockInterceptor` (version 1.2), that evolved into class `PessimisticLockInterceptor` in versions 1.4 and 2 (line 23). Refinement is also noticeable: an important difference between versions 1.4 and version 2.0 is the refinement of the implementation for transaction management, in particular the management of transactional contexts that has been refined using three new classes, see lines 13 to 15. Even though these changes present evidence of refactoring and refinement, a class list is not definite evidence for these kinds of evolution: below we will have a closer look on this modifications and show that the implementation is largely resilient to the improvement of their crosscutting characteristics.

3.3.2 Implementation structures and crosscutting

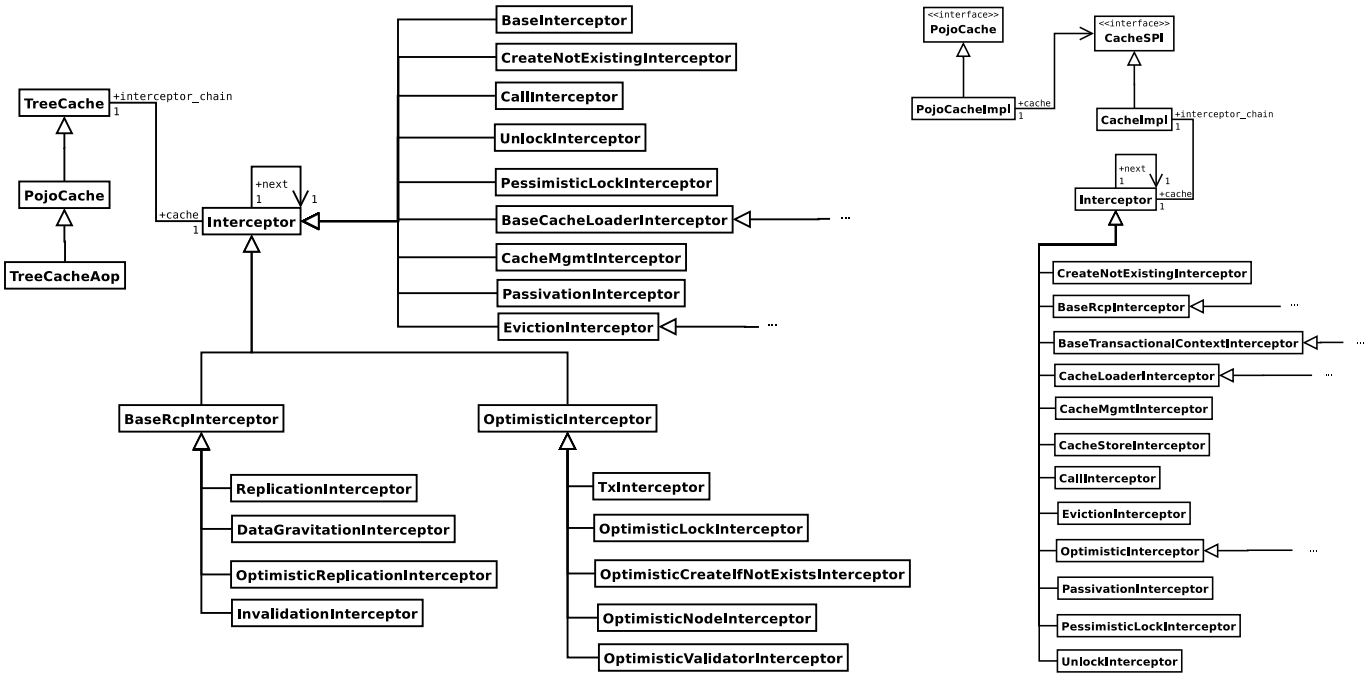
Figure 3.5 shows a more detailed view by means of class diagrams of the evolution of the main object-oriented implementation structures (*i.e.*, data structure and interceptors) in JBoss Cache (JBC) versions 1.4 and 2. In JBoss Cache version 1.4 (see figure 3.5a) the classes supporting the `TreeCache` data structure and the POJO caching functionalities are almost unchanged with respect to version 1.2 (see figure 3.4a): only one new class `PojoCache` appears in the inheritance hierarchy, this class replaces the class `TreeCacheAOP` from version 1.2 (see figure 3.4a); the class`TreeCacheAOP` is kept only as a wrapper to assure backward compatibility. JBoss Cache version 2 includes changes to the main structures, see figure 3.5b. In the latest version, the POJO caching functionality is structured as a hierarchy composed by the interface `PojoCache` and an implementing class, `PojoCacheImpl` (note that the abbreviation "AOP" has disappeared form the naming conventions). The cache now uses the classes implementing the caching functionality, *i.e.*, the interface `CacheSPI` and the class `CacheImpl`.

The hierarchy of interceptors has also been subject to notable changes since version 1.2. First, as we have mentioned before, the hierarchy has been extended and now contains interceptors supporting new functionality, *e.g.*, data gravitation (see class `DataGravitation-Interceptor` under `BaseRcpInterceptor` class hierarchy). Second, some code structures have been refactored, and we can see how they evolved in the hierarchy during evolution. For example, the interceptor for replication `ReplicationInterceptor` inherited directly from class `Interceptor` in version 1.2 (see figure 3.4a), however in versions 1.4 and 2 it is located under the hierarchy of the more general interceptor `BaseRcpInterceptor`, that supports a sub-hierarchy of specialized replication interceptors.

To complete this qualitative analysis of the JBoss Cache code structure we have performed a detailed statement-level analysis for the later versions as presented before for the first version, see figure 3.4b. Figure 3.6 illustrates the scattering of replication and transaction

		2.0.0	1.4.1	1.2.1
1	**Interceptor mechanism**			
2	Interceptor	x	x	x
3	BaseInterceptor		x	
4	CallInterceptor	x	x	x
5	CreateIfNotExistsInterceptor		x	x
6	**Replication**			
7	BaseRpcInterceptor	x	x	
8	DataGravitatorInterceptor	x	x	
9	InvalidationInterceptor	x	x	
10	OptimisticReplicationInterceptor	x	x	
11	ReplicationInterceptor	x	x	x
12	**Transactions**			
13	BaseTransactionalContextInterceptor	x		
14	InvocationContextInterceptor	x		
15	NotificationInterceptor	x		
16	TxInterceptor	x	x	
17	OrderedSynchronizationHandler	x	x	x
18	OptimisticInterceptor	x	x	
19	OptimisticCreateIfNotExistsInterceptor	x	x	
20	OptimisticLockingInterceptor	x	x	
21	OptimisticNodeInterceptor	x	x	
22	OptimisticValidatorInterceptor	x	x	
23	PessimisticLockInterceptor	x	x	LockInterceptor
24	UnlockInterceptor	x	x	x
25	**Cache Loading**			
26	CacheLoaderInterceptor	x	x	x
27	ActivationInterceptor	x	x	
28	BaseCacheLoaderInterceptor		x	
29	CacheStoreInterceptor	x	x	x
30	**Eviction**			
31	EvictionInterceptor	x	x	
32	**Pasivation**			
33	PassivationInterceptor	x	x	
34	**Management and JMX**			
35	Several classes	10 Classes	9 Classes	

Table 3.1: List of interceptors classes in the three versions of JBoss Cache classified according to the main features



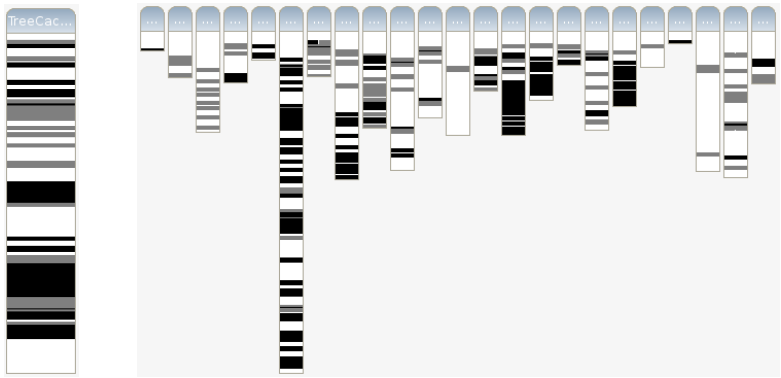
(a) JBC 1.4.1 (b) JBC 2.0.0

Figure 3.5: Evolution of code structure in JBoss Cache

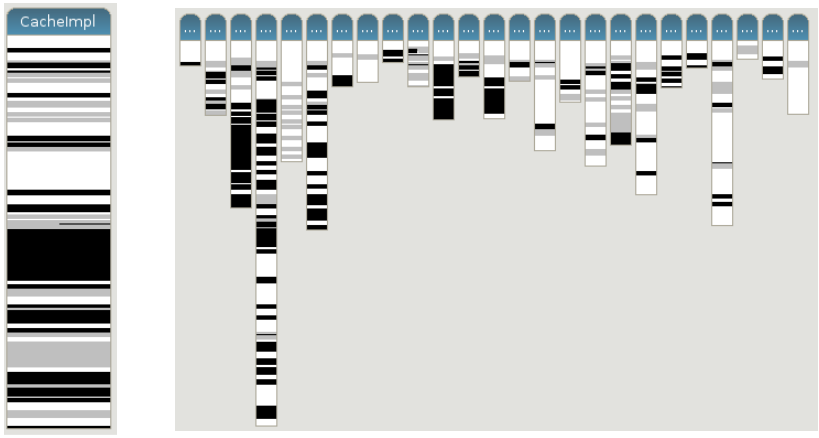
code in versions 1.4 and 2 of JBoss Cache. The main data structure class are represented by the left column in the figures, and the interceptor packages are represented on the right. As before, replication code is colored gray in each subfigure, and transaction code is marked black. The figures clearly exhibit scattering and tangling of replication and transactional code with respect to one another and with respect to functional code of the cache, as exhibited before for version 1.2. Furthermore, the figures show that the tangled code has not diminished over the two evolution steps and, even worse, it seems that the scattering phenomena has augmented. To sum up, these results show that a clean modularization of the replication and transaction functionalities of JBoss Cache has not been achieved through two major evolution steps, even though evolution has partially explicitly been targeted towards addressing modularization issues of these functionalities.

3.3.3 Quantitative analysis of crosscutting concerns

To complete our analysis and provide definite evidence to the claim that crosscutting has prevailed over the evolution of JBoss Cache, table 3.2 presents metrics regarding the modularization of three crosscutting concerns: replication, transactional behavior and the calls to/from the interceptor package. For example, the `TreeCache` class, the main data structure in JBoss Cache version 1.2, comprises 188 methods and consists of 1741 lines of code (LOC): the scattered code relevant for replication amounts to more than 196 LOC; the code for transactions accounts for more than 228 LOC. The situation for the interceptor framework is similar: it includes nine classes consisting of 1263 LOC altogether; the (lower-bound) line counts for code relevant to replication, transactions and calls to `TreeCache` respectively are



(a) JBC 1.4.1



(b) JBC 2.0.0

Figure 3.6: Entangled and scattered code for distribution (gray) and transactions (black) in the evolution of JBoss Cache. On the left of each figure the main class implementing the tree data structure, and on the right side the interceptors package.

	JBoss Cache version		
	1.2.1	1.4	2.0.0
Main Design structure			
LOC for replication	196	74	55
LOC for Transactions	228	280	197
Total LOC	1741	3802	2899
Interceptor Package			
Number of classes	9	33	34
LOC for replication	30	123	121
LOC for Transactions	41	137	274
LOC for calls to the main structure (*i.e.*, class representing the cache)	73	165	132
Total LOC	1263	5099	5219

Table 3.2: Metrics evolution in the refactoring process of JBoss Cache

30, 41, and 73.

As to version 2, from the 2899 lines of code (LOC) of class `CacheImpl`, the analogue to `TreeCache` in version 1.2, the scattered code relevant for replication amounts to more than 55 LOC and the code for transactions accounts for more than 197 LOC. The situation of the interceptor framework is similar, complexity and scattering of code is increasing. The package includes 34 classes consisting of 5219 LOC altogether; the (lower-bound) line counts for code relevant for replication, transactions and calls to `Cache` (`CacheImpl` interface) respectively are 121, 274, and 132. This provides strong evidence that the interceptor mechanism is not sufficient to modularize crosscutting concerns and that continuous refactoring does not reduce or simplify the complexity problem due to crosscutting concerns.

3.4 Discussion

This section has shown that understanding the replication and transactional behavior of the JBoss Cache implementation, which uses the interception mechanism (*i.e.*, filter pattern), is far from trivial. It also shows that continuous evolution that implies far-reaching re-engineering and refactoring actions has not resulted in a better separation of concerns in the evolved code. This crosscutting code results in difficulties for the extension of the cache, for example, by even simple new replication policies. This applies, *e.g.*, if the replication policy should be modified to one replicating only when a cache is interested in some specific data and only within the subgroup of hosts that are also interested in the same data instead of replicating always between all members of a cluster (Note that this is a dynamic caching behavior, different from the buddy replication, or the region support proposed in the newest versions of JBoss Cache). Finally, note that traditional AspectJ-like languages are not appropriate in this context: as shown by Nishizawa et al. [NST04] and Soares et al. [SLB02] they are subject to limitations, in particular, requiring inadequately complex aspect definitions in the context of distributed crosscutting functionalities.

Chapter 4

The AWED language

Modularization of crosscutting concerns for distributed applications using an aspect language, *i.e.*, in terms of pointcut, advice and aspect abstractions, suggests support for the following issues: (i) a notion of remote pointcuts allowing to capture relationships between execution events occurring on different hosts, (ii) a notion of groups of hosts which can be referred to in pointcuts and manipulated in advice, (iii) execution of advice on different hosts in an asynchronous or synchronous way and (iv) flexible deployment, instantiation, and state sharing models for distributed aspects.

AWED provides such support through three key concepts at the language level. First, *remote pointcuts*, which enable matching of join points on remote hosts and include remote calls and remote cflow constructs (*i.e.*, matching of nested calls over different machines). As an extension of previous approaches AWED supports remote regular sequences as other approaches did for (non-distributed) regular sequence pointcuts [DFS02, DFS04, DFL$^+$05, A$^+$05, VSCDF05]. Second, support for *distributed advice*: advice can be executed in an asynchronous or synchronous fashion on remote hosts and pointcuts can predicate on where advice is executed. Third, *distributed aspects*, which enable aspects to be configured using different deployment and instantiation strategies. Furthermore, aspect state can be shared flexibly among aspect instances, as well as among sequence instances which are part of an aspect.

Apart from these conceptual considerations AWED design considers and addresses four insufficiencies found in the approaches described in chapter 2. First, we avoid technology rigidity, making an extensible language which can be connected to different technologies for distribution, *e.g.*, Java RMI, JGroups, J2EE, CORBA, using adapters. Second, we do not have a centralized architecture, instead AWED supports a decentralized architecture with dynamic management of new hosts. Third, we provide language support with an extensible library, combining the advantages of language with the flexibility of libraries. Finally, we provide explicit management of groups of hosts as a mechanism for more expressivity of distributed predicates.

4.1 AWED as a distributed aspect language

AWED is an aspect language for distribution, therefore it can be classified according to the taxonomy presented in section 2.1. We use such classification as a guide to detail the main hypothesis and restrictions influencing AWED design. Table 4.1 summarizes AWED features

according to the taxonomy.

Taxonomy elements	Values
Communication model	
Communication mechanisms	
Remote method call	yes
Join point propagation	yes
Controlled remote advice invocation	yes
Group communication	yes
Parameter passing modes	by reference and by copy
Synchronization model	
Synchronization mechanisms	synchronous control, non-blocking, or futures
Communication timing	synchronous, causaly ordered asynchronous FIFO
Asynchronous hypothesis	yes
Causal predicates	yes
Remote pointcut model	
Expressiveness:	
Atomic	yes
Sequential control flow	yes
Distributed control flow	yes
History based	yes
Finite-state	yes
Vpa	no
Context-free	no
Turing-complete	yes
Remote join point support	
All pointcuts	yes
Location	Single host, Multiple host groups
Paradigm:	
Object-oriented	yes
Functional	no
Logic	no
Remote advice model	
Filtering and ordering	yes
Location	Single host, Multiple host groups
Synchronization mode	Synchronous and asynchronous
Parameter passing modes	by reference and by copy
Parameter passing mechanism	tag language
Proceed support	yes
Remote host only	yes
Remote and originating host	yes
Synchronization mechanisms	Blocking and transparent futures
Mobility	weak
Reflective access to program state	yes
Aspect model	
Instantiation	Declarative
Declarative	
Per thread	yes
Per class	yes
Per object	yes
singleton	yes
Per cflow	yes
Per binding	yes
Deployment	Dynamic
Deployment scope	global,local, group of hosts, and remote host
State sharing	global, local, group
Weaving mechanisms	load-time
Aspect composition	
Type	Explicit
Mechanisms	Precedence
Object	Advice
Scope	All
Expressiveness	Order
Distributed guarantee	yes

Table 4.1: Taxonomy for distributed aspect-based systems

AWED is designed as an extension of Java and as such shares some of the features provided by Java. First, AWED relies on processes and threads as abstractions for activities. A process is an independent execution space, that do no shares memory with others processes, and where many threads may be running at the same time. During this discussion we refer to these processes as hosts. Even though, processes are independent units with no shared memory, AWED allows to define specific state variables inside aspects to be shared among groups of hosts. Regarding communication, AWED relies on remote procedure call, as a super set of remote method invocation, that includes remote advice invocation (see issue remote method call on table 4.1). For such invocations AWED supports object passing by-copy and by-reference (issue parameter passing mode). To address synchronization between

asynchronous processes AWED provides transparent futures [RHH85]. For the sake of scalability, low coupling, and flexibility, communication may be synchronous, causally ordered, and asynchronous with FIFO (first in first out) guarantee (issue communication timing). Note that FIFO guarantee is only provided to messages coming from the same host (*i.e.*, process). The causality support builds conceptually from AWED's design as system with asynchronous FIFO support (where as demonstrated by Charron-Bost, Mattern, and Tel [CBMT96] the following hierarchy holds: $Synchronous \subset Causally\ Ordered \subset Asynchronous\ FIFO$ [1]).

AWED provides a pointcut model based on remote pointcuts, *i.e.*, all pointcuts have distributed semantics. Therefore, the atomic pointcuts provided by AWED match atomic events (*e.g.*, method calls) on any participating host unless they are restricted with location pointcuts (see issue remote join point support and expressiveness). Additionally to atomic pointcuts AWED also supports history based pointcuts (issue history based). For example, it can match join points that spawn over several host in the same control flow. Furthermore, it supports regular aspects that match patterns of join points defined by means of a finite state machines. Note that pointcuts semantics are based on object oriented abstractions.

The remote advice model provides synchronous and asynchronous advice execution, filtering and ordering of remote advice execution, explicit location of advice execution inside group of hosts, and weak mobility for aspects (*i.e.*, aspects definitions are copied to remote hosts). Note that composition of aspects is determined by the aspect precedence, that is controlled by the originating host, *i.e.*, the host where the join points occurs. Note also that AWED provides declarative means to deal with instantiation of aspects. Finally, AWED supports asynchronous systems and do not impose a global clock as an hypothesis in the model (*i.e.*, no global clock).

We now analyze the syntax and semantics of the resulting language.

4.2 Syntax and semantics

AWED's syntax is shown in Fig. 4.1 using EBNF formalism (*i.e.*, square brackets express optionality; brackets denote multiple occurrences, possibly none; terminal parentheses are enclosed in apostrophes).

4.2.1 Pointcuts

AWED employs a model where, upon occurrence of a join point, pointcuts are evaluated on all hosts where the corresponding aspects are deployed. Thus, in AWED all join points are remote and in principle and each join point is distributed to the hosts that are interested on it (*i.e.*, hosts where an aspect that may match the join point is deployed). However, not all possible join points are distributed (*i.e.*, all method calls). Instead, AWED uses pointcut definitions to decide what specific join points may be matched and then distributed. For

[1] The $\subset$ operator represents the subset relation, implying for example that *Synchronous* systems are a subset of *Causally Ordered* systems. In [CBMT96] the authors consider a distributed systems as a set of n processes, communicating only by message exchange, and without any assumption over the processors speed or message propagation delay. In such a model the authors formally define a distributed computation as a n-tuple $C = (C_1, ..., C_n)$ of local computations C_i (where each local computation represent the sequence of events on each process), and a set of pairs representing the communication messages $ComMsg = \{(s, r) \in C_i \times C_j : s\ and\ r\ are\ the\ send\ and\ receive\ events\ of\ a\ message\}$. Using this model authors define several types of computations, *e.g.*, synchronous computations, and are able to prove several properties (*e.g.*, the one presented in the document). Further information can be found in [CBMT96].

// Pointcuts

Pc ::= `call`(*MSig*) | `execution`(*MSig*)
| `get`(*FSig*) | `set`(*FSig*)
| `cflow`(*Pc*) | *Seq*
| `host`(*Group*) | `on`(*Group*[, *Select*])
| `target`({*Type*}) | `args`({*Arg*})
| `eq`(*JExp*, *JExp*) | `if`(JExp)
| `within`(*Type*)
| `passbyval`({*Id*})
| *Pc* || *Pc* | *Pc* && *Pc* | !*Pc*

Seq ::= [*Id*:] `seq`({*Step*}) | `step`(*Id*,*Id*)

Step ::= [*Id*:] *Pc* [→*Target*] | `start`: →*Target*

Target ::= *Id* | *Id* || *Target*

Group ::= { *Hosts* }

Hosts ::= `localhost` | `jphost` | "*Ip*:*Port*"
| *GroupId*

GroupId ::= *String*

Select ::= *JClass*

// Advice

Ad ::= [`asyncex`] *Pos*({*Par*}) : *PcAppl* '{' {*Body*} '}'

Pos ::= `before` | `after` | `around`

PcAppl ::= *Id*({*Par*})

Body ::= *JStmt* | `proceed`({Arg}) | `localproceed`({Arg})
| `addGroup`(Group) | `removeGroup`(Group)

// Aspects

Asp ::= [*Depl*] [*Inst*] [*Shar*] `aspect` *Id* '{' {*Decl*} '}'

Depl ::= `single` | `all`

Inst ::= `perthread` | `perobject` | `perclass`
| `perbinding`

Shar ::= `local` | `global` | `inst` | `group`(*Group*)

Decl ::= [*Shar*] *JVarD* | *PcDecl* | *Ad*

PcDecl ::= `pointcut` *Id*({Par}) : *Pc*

// Standard rules (intensionally defined)

MSig, *FSig* ::= *// method, field signatures (AspectJ-style)*

Type ::= *// type expressions*

Arg, *Par* ::= *// argument, parameter expressions (AspectJ-style)*

Id ::= *// identifier*

Ip, *Port* ::= *// integer expressions*

JClass ::= *// Java class name*

JExp ::= *// Java expressions*

JStmt ::= *// Java statement*

JVarD ::= *// Java variable declaration*

Figure 4.1: AWED language

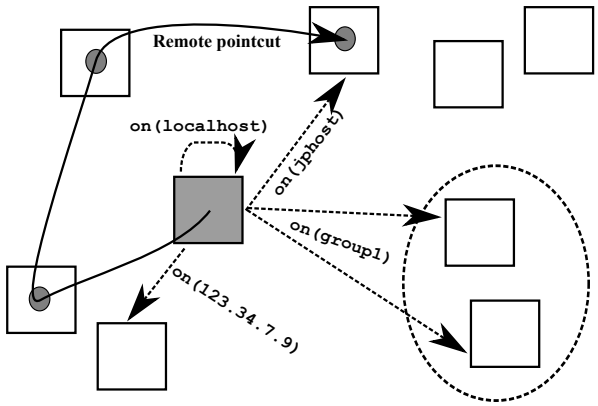


Figure 4.2: Remote pointcuts and advice in AWED

example, consider a graphical application that contains an excerpt of code that calls several methods:

```
aCircle.setCenter(5.0, 7.8);
aCircle.setRadius(3.7);
aCircle.draw();
```

A programmer implementing a system to reproduce the drawing in host A into host B may write a pointcut to match (*i.e.*, to detect) a call to the method `draw` in host A. In such case, an event signaling that a specific host has reached a call to the method `draw` will be distributed to host B.

Pointcuts (which are generated by the non-terminal *Pc*) are basically built from `call` constructors (`execution` allows to denote the execution of the method body), field getters and setters, nested calls (`cflow`) and sequences of calls (non-terminal *Seq*). Pointcuts may then contain conditions about (groups of) hosts where join points originate (term `host`(*Group*)), *i.e.*, where calls or field accesses occur. Furthermore, pointcuts may be defined in terms of where advice is executed (term `on`(*Group*)). Advice execution predicates may further specify a class implementing a selection strategy (using the term `on`(*Group*, *Select*)) which may act as an additional filter or define an order in which the advice is executed on the different hosts. Groups are sets of hosts which may be constructed using the host specifications `localhost`, `jphost` and `adr:port`, which respectively denote the host where a pointcut matches, the host where the corresponding join point occurred and any specific host. Alternatively, groups may be referred to by name. (Named groups are managed dynamically within advice by adding and removing the host which an aspect is located on, see Sec. 4.2.4 below.)

Finally, pointcut definitions may extract information about the executing object (`target`) and arguments (`args`). They may also test for equality of expressions (`eq`), the satisfaction of general conditions (`if`), and whether the pointcut lexically belongs to a given type (`within`). Pointcuts may also be combined using common logical operators.

Figure 4.2 illustrates the model for remote pointcuts and advice. Pointcuts essentially allow to match sequences of execution events that occur on different hosts. Hosts can be referred to using absolute addresses but can also be defined relative to the host on which an aspect is deployed (term `localhost`, in the figure the host colored in gray). Remote advice can be triggered on other hosts using the `on` specifier. Besides the host specifications available for pointcut definitions, advice execution can also be specified to take place on the host where

the pointcut has been matched (term `jphost`). Pointcuts and remote advice execution may depend on explicitly defined groups of hosts. In pointcuts, such groups may limit matching of execution events to sets of hosts; as to advice executions, groups allow to execute advice on several hosts. Furthermore, AWED allows to execute pieces of advice synchronously or asynchronously with the execution of the base application and with other aspects.

As a first example, the following simple pointcut could be part of a replicated cache aspect:

call(**void** initCache()) && **host**("192.168.0.1:5678")

Here, the pointcut matches calls to the cache's initCache method that originate from the host that has the specified address. The advice will be executed on any host where the aspect is deployed (possibly multiple ones) as there is no restriction on the advice execution host. The following example restricts the execution hosts to be different from the host where the joinpoint occurred:

pointcut putCache(Object key, Object o):
call(∗ Cache.put(Object,Object)) && !**on**(**jphost**) && **args**(key, o)

Here, the first line defines the pointcut `putCache` with two arguments of type `Object key` and `o`. Then the pointcut definition defines a pointcut that matches calls to the cache's put operation on hosts other than the joinpoint host and binds the corresponding data items: first, a `call` pointcut is defined to match methods calls to method `put` on objects of type `Cache` with any returning value. Then, the `on` pointcut is defined to execute the advice in hosts that are not the host where the pointcut was originated, *i.e.*, not in the hot of the join point (!`on`(`jphost`)). Finally, using the `args` pointcut the code binds the first argument of matched method to the first parameter of pointcut definition (argument `key`), and the second argument of the matched method to the second argument of the pointcut (argument `o`). Note that in this case the clause !`host`(`localhost`) could replace !`on`(`jphost`) to achieve exactly the same effect of matching non-local joinpoints. If the corresponding advice puts the item in the local cache, a condition on the aspect type (named, *e.g.*, `ReplCache`), such as !`within`(`ReplCache`), can be used to avoid triggering the pointcut during the advice execution.

4.2.2 Sequences

Sequences (derived by the non-terminal *Seq* in figure 4.1) are supported by two constructions on the pointcut level. First, the term [*Id*:] `seq`({*Step*}) allows to define sequences which may be named using an identifier and consist of a list of (potentially named) steps (non-terminal *Step*). A step may define the steps to be executed next (non-terminal *Target*).[2] A sequence matches if the sequence is completed, *i.e.*, if the current joinpoint matches the last step and previous joinpoints of the execution trace matched the previous steps in order. Second, the term `step`(*seq*,*step*) matches if the step named *step* of the sequence named *seq* occurs. This allows advice to be triggered after a specific step within a sequence using a term of the form s: `seq`(... l: logout() ...) && `step`(s, l).

To illustrate the use of sequence pointcuts, consider the following pointcut, which could be part of a simple cache replication protocol:

pointcut replPolicy(Cache c):

[2]Note that while our sequences obviously encode finite-state automata, many applications of regular structures, in particular communication protocols [DFL+05], are effectively sequence-like, *i.e.*, of a one-dimensional directed structure, so that we decided to use the more intuitive terminology for AWED.

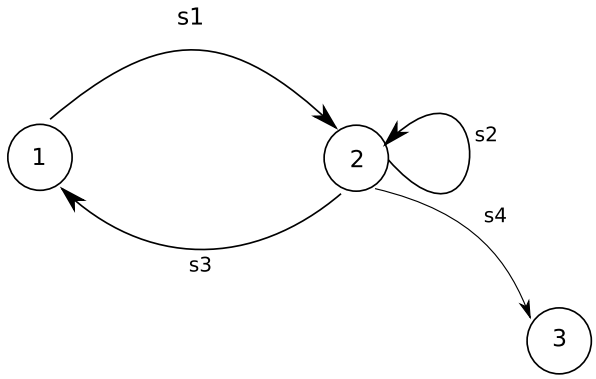


Figure 4.3: Automaton representing first sequence example

replS: **seq**(s1: initCache() && **target**(c) > s3 || s2 || s4,
s2: cachePut() > s3 || s2 || s4,
s3: stopCache() > s1,
s4: cacheInconsistency())

(Here, identifiers like initCache denote undefined pointcuts specifying corresponding call pointcuts.) The pointcut above defines a sequence of four steps. An initialization step which may be followed either by a put operation (s2), termination of the cache (s3) or an error step (s4). A put operation (s2) may be repeated, followed by a cache termination event (s3) or resulting in a cache inconsistency (s4). After cache termination, the cache may be initialized once again. Finally, a cache inconsistency terminates the sequence (and may be handled by advising the pointcut). Figure 4.3 shows a graphical representation of this automaton, in the automaton each state is defined according to the transitions (steps) that it accepts. Additionally, if the initial state is not specified in the definition (term `start`), the first state accepts only the first defined transition by default.

Note that the above definition does not enforce that the steps are taking place in the context of the same cache. This is, however, simple to achieve by binding the targets of the different steps using `target`(c), and use `eq` or `if` pointcuts to ensure the appropriate relationships at different steps in a sequence.

A step may be referred to in pointcuts and advice as exemplified in the following example:

pointcut putVal(Cache c, String key, Object o):
step(replS, s2) && **target**(c) && **args**(key, o)

which shows how to provide a special pointcut for the second step in the previous sequence (and how to bind the variables used in that step) so that advice can later be attached to it.

4.2.3 Parameter passing

Sharing of remote object information is an inherent need in distributed applications. AWED's proposal includes a mechanism to manage explicitly how parameters of a given joinpoint are distributed.

The pointcut language model allows joinpoint information like parameters, the caller object, and the target object to be bound to specific variables in pointcut or advice definitions. The model also allows those variables to be explicitly distributed by value or by reference (the latter being the default behavior). The first option, by value, creates a copy of the object in

```
1 pointcut myDef(Integer i, Object y, Vector v, MyObject c):
2     call(* foo(Integer,Object, Vector, String)) &&
3     args(i, y, v, String) && target(c) &&
4     passbyval(i, c);
```

Figure 4.4: Parameter passing example using a pointcut definition

the hosts where remote advice are executed and binds the new value to the formal parameter used in the advice body. The second option, by reference, creates a remote reference to the original object. Once the remote reference or the copy are bound, they can be treated as local objects without any distinction in the advice body. Note that we could have defined the pass by value term as a modifier of the parameters of the `execution` or `call` pointcuts. However, to avoid confusion with the matching semantics of those constructs, and to allow reuse of pointcut definitions with different parameter passing policies we reify parameter passing as an independent pointcut construct.

Figure 4.4 shows an example of the usage of the pass by value behavior. The pointcut definition has four variables `i, y, v` and `c`. The parameters of the matched method, `foo`, and the target object of that method are bound to the pointcut's variables. The term `passbyval` is used to annotate the variables `i` and `c` to be passed by value.

As usual, by value passing has to be used with care, in particular, because it implies copying the whole object graph below the object that is passed by value. AWED allows all objects to be referenced remotely, copied and distributed. Reflective information queried through the `thisJoinPoint` keyword is always passed by reference. For instance, when invoking `getTarget()` method in the `thisJoinPoint` object inside an advice, a remote reference to the target is returned. This is done allow programmers to access the original objects, if they are interested only in a copy they can use the passing mechanisms to bind the copies to specific arguments in the pointcut definition.

The remote reference model of AWED is fully transparent with respect to the object model. This means that there is no distinction between the remotely referenced objects and the locally referenced objects. However, it is important to note that the language provides a richer and finer grained model for parameter passing in the pointcut definition language than the one provided in direct method invocation over objects. When a method is invoked directly in a remote referenced object, the parameters are passed by value as with normal Java method invocations. There is no language support to specify pass-by-reference behavior (although a work-around using a proxy object is possible). This is motivated by the fact that we aim to stay as close as possible to Java and changing the method invocation syntax and semantics would be a drastic departure of this principle.

4.2.4 Advice

Advice (non-terminal *Ad* in Fig. 4.1) is mostly defined as in AspectJ: it specifies the position (*Pos*) where it is applied relative to the matched join point, a pointcut (*PcAppl*) which triggers the advice, a body (*Body*) constructed from Java statements, and the special statement `proceed` (which enables the original call to be executed).

In an environment where advice may be executed on other hosts (which is possible in AWED using the `on` pointcut specifier), the question of synchronization of advice execution with respect to the base application and other aspects arises. AWED proposes one

unified model for (local and remote) advice execution: all advice (including remote ones) are triggered by one controller. This means that there is only one advice chain per joinpoint. The host where the joinpoint occurs is responsible for managing the advice chain. As such, there is a well-defined precedence as defined by the AspectJ precedence rules (e.g. `declareprecedence`), even for advice executed on remote hosts.

Per default, advice executes synchronously to the base application, meaning that the application waits until completion of the advice in order to proceed to the original behavior or to execute the next advice in the chain. Both remote and local advice conform to this general model. The programmer may also choose to execute an advice asynchronously with respect to the application on the joinpoint host by marking the advice with the `asyncex` keyword. This means that the base application proceeds with the original behavior or executes the next advice in the chain while the asynchronous advice is executing. Of course, asynchronous advice is still treated in the same advice chain and thus are only executed when the previous synchronous advice are finished or when previous asynchronous advice are started.

Multiple around advice applying at a joint point is executed as usual in a nested fashion as part of a chain controlled by the invocation of `proceed` (see figure 4.5). Such advice is expected to return a value that can be processed by the previous advice that invoked proceed or by the original application in case of the first advice. In case of asynchronous advice, this value is possibly not yet computed, so the invocation of such an advice returns a future[3] object [RHH85] that synchronizes with the remote advice in case the object is claimed. The future object is implicit, as such the advice caller can safely treat the return value as an actual value. The AWED infrastructure and run-time weaver take care of generating a transparent future and claiming it whenever its value might be accessed. It is also possible to make the future explicit by casting it to a standard Java Future object[4] when useful. This way, the invoking advice may manage its behavior depending on the availability of the result.

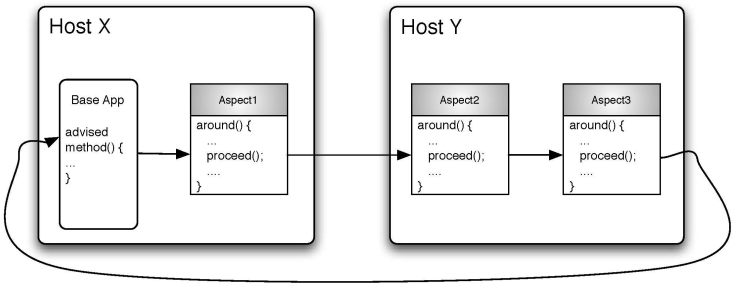


Figure 4.5: Around advice chaining in AWED. Advice applicable to the same joinpoint execute in a single advice chain, regardless of execution host.

AWED introduces one general model for (a)synchronous distributed advice. The semantics of AWED remains backward compatible with AspectJ. The semantics of advice is also independent of whether the joinpoint host is different or the same as the execution host. This is in contrast to the previous version of AWED [BNSV+06a], where advice was treated dif-

[3] A future object is an object that represents the result of an asynchronous computation, the actual value of the computation is bound to the object once the concurrent computation is finished. This concept was first presented in MULTILISP [RHH85].

[4] Java supports futures since version 1.5 through the `java.util.concurrent` API.

```
1 around(User user,Object ttarget): toAuthenticate(user,ttarget) {
2     if(userMayAccess(user,ttarget))
3        proceed();
4     else throw new AccessSecurityException(user,ttarget);
5 }
```

Figure 4.6: Simple authentication around advice

```
1 pointcut distribution(Facade f):
2   target(f) && call(* *(..)) &&
3     && !host("Serveripadr:port") && on("Serveripadr:port");
4
5 syncex Object around(Facade f): distribution(f) {
6   return localproceed(Facade.getInstance()); }
```

Figure 4.7: Distribution as an aspect

ferently because every host had its own advice chain. In that model, advice always executes asynchronously to advice on different hosts, impeding, for instance, remote authentication that blocks all other advice and the original behavior until authentication has been successful. The new general model of AWED is able to support such behavior as illustrated by the code fragment of figure 4.6. The authentication advice is guaranteed to always execute before other advice that applies to that joinpoint such as, for instance, a billing advice [DJ04]. The billing advice will not be executed when the authentication fails because the authentication advice does not invoke `proceed` in that case.

The advice body has one important additional keyword in comparison to AspectJ: `localproceed`. The invocation of `localproceed` makes sure that the original behavior (*i.e.*, the joinpoint) is executed on the host where the advice occurs instead of on the host where the original behavior originated from (*i.e.*, the joinpoint host). As an example, consider the aspect shown in Fig. 4.7 that implements the distribution concern. It is well-known that distribution can be seen as a crosscutting concern that can be modularized using aspects (see, *e.g.*, [SLB02]). The `distribution` pointcut selects all calls to `Facade` methods on the client and makes sure that the accompanying advice is only executed at the server side. By employing the negation of the `host` designator, calls on the server side will not match the pointcut themselves. The redirection behavior is encapsulated in a synchronous around advice. As the around advice gets executed on the server host, the `getInstance` method of the `Facade` class will retrieve an instance which is local to the server host The `localproceed` expression makes sure to invoke the original behavior on the server host instead of that on the joinpoint host. An interesting variation of the distribution concern would be to mark the advice asynchronous. The result is that the original sequential application is not only distributed but also parallelized. In such a case, a future object is immediately returned to the base application and the advice is executed in parallel with the base program. The base program and the advice get synchronized once the actual value of the computation is claimed through the future.

Finally, note that AWED enables advice to manage named groups of hosts: `addGroup` adds the current host to a given group, `removeGroup` allows to remove the current host from a group.

4.2.5 Aspects

Aspects (non-terminal *Asp*) are composed of a set of fields as well as pointcut and advice declarations. Aspects may be dynamically deployed (*Depl*) on all hosts (term `all`) or only the local one (term `single`).

Furthermore, aspects support four instantiation modes (*Inst*): similar to several other aspect languages, aspects may be instantiated per thread, per object, or per class. However, aspect instances may also be created for different sets variable bindings arising from sequences (term `perbinding`) as introduced in [DFL+05, A+05]. In this last case, a new instance is created for each distinct set of bindings of the variables in the sequence, *i.e.*, of the variables declared as arguments of a sequence pointcut or fields used in the sequence pointcut [5].

Finally, AWED allows distributed aspects of the same type to share local state (*Shar*): values of aspect fields may be shared among all aspects of the same type (term `global`), all instances of an aspect which have been created using the instantiation mechanisms introduced before (non terminal *Inst*), all aspects belonging to the same group (term `group`(*Group*)) or all aspects on the one host (term `local`; note that these possibly belong to different execution environments, such as JVMs). Sharing modifiers can be given for an aspect as a whole or individual fields (*Decl*), if both are given, the latter have priority.

4.3 AWED by example

In this section, several applications of AWED are presented to show the basic capabilities of the language:

- First, we illustrate the usage of some of the basic mechanisms for explicit distribution in AWED: `host` and `on` pointcuts, and a synchronous around advice. In particular we show how distribution can me introduced into a non-distributed application (Sec. 4.3.1),
- we then extent such distribution example showing how execution clustering concerns can be introduced concisely using groups and remote execution policies (Sec. 4.3.2).
- Finally, in section 4.3.3 we illustrate a more complex example showing how replicated, cooperating distributed caches can be modularly implemented.

4.3.1 Distribution

In [SLB02], Soares *et al.* illustrate how AOP techniques can be employed to explicitly introduce distribution within existing, non-distributed applications. For this, AspectJ is employed to automatically insert the required RMI code fragments. Their proposal requires two types of aspects: one aspect for handling server distribution concerns and one aspect for handling client distribution concerns. At the server side, a `Remote` interface is statically introduced for each object that should be remotely available. Additional methods are introduced by means of the server aspect to export and manage the remote objects. The client side aspect is responsible for capturing and redirecting the local method calls and declaring these methods to throw remote exceptions. In addition, each method specified in the remote interface requires a dedicated redirection advice, as in RMI the type of remote objects is the remote interface type and not the actual object type, and as such, AspectJ does not allow to change

[5] The current prototype implementation of AWED does not include `perbinding` instantiation support.

```
1 pointcut distribution(Facade f):
2   target(f) && call(* *(..)) &&
3     && !host("Serveripadr:port") && on("Serveripadr:port");
4
5 Object around(Facade f): distribution(f) {
6   return localproceed(Facade.getInstance()); }
```

Figure 4.8: Distribution as an aspect

the target object by another of different type in a *proceed* statement, which is required to redirect the calls to the remote objects. Additionally, AspectJ is used to implement the mechanism to handle synchronization of objects passed remotely by copy (default behavior in RMI for method's parameters and method's returning objects that do not implement the `Remote` interface) and their remote pairs.

AWED allows for a more elegant solution, which does not require the overhead of introducing the required RMI-specific code. AWED allows to solve this distribution problem using a single aspect which is illustrated in figure 4.8. The `distribution` pointcut selects all calls to `Facade` methods on the client and makes sure that the accompanying advice is only executed at the server side. By employing negation of the host designator, calls on the server side will not match the pointcut themselves. The redirection behavior is encapsulated in a synchronous around advice. As the around advice gets executed on the server host, the `getInstance` method of the `Facade` class will retrieve an instance which is local to the server host (this could be generalized in order not to rely on a single object, the singleton idiom is used for simplicity and could be replaced by a naming mechanism, *e.g.*, Java Naming and Directory Interface (JNDI) [Myc08]). The `proceed` expression will invoke the original behavior on that Facade instance located on the server host. The AWED solution improves on the AspectJ-based solution: first, there is no need for RMI specific code to be injected in the server classes, which is a tedious process, and secondly, only one aspect with one pointcut and advice suffices while in the AspectJ solution at least two aspects and a pointcut-advice pairs for each method in the Facade class are necessary. Note that AWED shares this advantage with other middleware-based AOP approaches, such as DJCutter [NST04] (Sec. 2.4.1), DAOP [PFFT02], and DYMAC [LJ06] (Sec. 2.3.2).

4.3.2 Clustering

When multiple servers are available to handle remote requests, one can choose to cluster these servers together such that incoming requests can be dispersed, *e.g.*, to balance the server load. Again, AWED provides an elegant solution and allows this clustering behavior to be encapsulated in a single aspect. Figure 4.9 illustrates this clustering pointcut. All available servers are part of the `ServerGroup` and the `on` designator specifies that the accompanying advice should be executed only on a server that is part of that specific group. As only one specific host should be the target of the redirection, a *Round Robin* load balancing mechanism is employed, which assigns a server host on a rotating base.

```
1 pointcut clustering(Facade f):
2   target(f) && call(* *(..)) && !host("Servergroup")
3     && on("Servergroup", awed.hostselection.RoundRobin);
4
5 Object around(Facade f): clustering(f) {
6   return proceed(Facade.getInstance()); }
```

Figure 4.9: Clustering as an aspect

```
1 all aspect CacheReplication{
2    pointcut cachePcut(Object key, Object o):
3     call(* Cache.put(Object,Object))
4         && args(key,o) && !on(jphost) &&
5     !within(CacheReplication);
6
7    before(Object key, Object o): cachePcut(key,o){
8      Cache.getInstance().put(key, o); }
9 }
```

Figure 4.10: Cache replication as an aspect

4.3.3 Caching revisited

n chapter 3 we have presented distributed caching and, in particular, its support through the JBoss Cache OO framework, as a motivating example for crosscutting in distributed applications. This section shows two different strategies of replication for distributed caches.

Simple strategy

Fig. 4.10 shows how an aspect for cache replication can be implemented using AWED. This aspect accounts for all places where cache elements are requested and replicated to all other caches in a cluster, *i.e.*, an essential part of the functionality of replication within JBoss Cache's `TreeCache` class.

The aspect declaration in line 1 indicates that the aspect will be distributed globally and that a singleton instance (AWED's default instantiation mode) is created on each host. The pointcut defined in lines 2–5 matches calls putting elements in the cache; the term `!on(jphost)` limits advice execution to aspects which are not deployed on the host where the join point matched. The advice (lines 7–8) simply puts the element in the cache. As the pointcut ensures that only aspects which are remote to the matching join point perform this advice, replication is thus achieved.

Adaptive summary-based strategy

As a more intricate example, we consider an example of the large number of replication strategies for caches that use hierarchical, cooperative and adaptive caching strategies [BO00, ICP]. Such strategies typically do not distribute data over whole clusters but replicate objects only to caches in the cluster that explicitly request them. Furthermore, cooperative behavior is useful, *e.g.*, looking for a copy in neighboring caches before (slowly) accessing farther caches or a centralized server holding the master copy of the data at hand. This kind of behavior

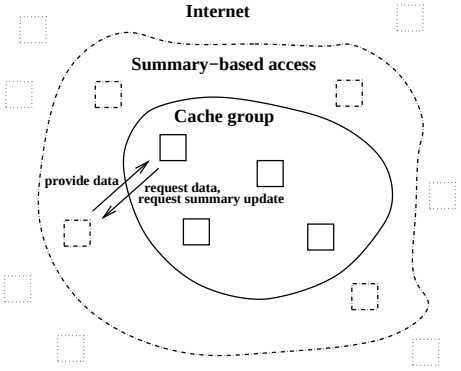


Figure 4.11: Adaptive cache behavior

is not part of the current JBoss Cache specification and would be very difficult to graft on its implementation without the use of AO techniques due to the crosscutting problems of its replication code.

In the following we present the heart of a summary-based cooperative cache strategy using AWED. Such caching strategies (see, *e.g.*, [FCAB00]) use "summaries", *i.e.*, small digests of the cache contents of neighboring caches. The summaries can be used to test whether a cache contains a value with high probability. They can therefore be used to guide the decision which neighboring caches to contact and thus reduce network traffic.In particular, a cache that receives a query for a specific value will first look locally for that value, if the value is not present it will check in the summary for some remote caches containing the value, and then it will ask for the specific value to the caches that may contain it. A instance of such case are web-proxies and web servers. There a proxy that receives a request for a specific web page will look for the page locally, and then, if the page is not stored locally, it will check in a summary list if other proxies have the specific page. If one of the other proxies has the page the proxy will ask for it. If non of the proxies has the page the request will be redirected to the web server.

Fig. 4.11 schematically illustrates a summary-based caching strategy used in the context of a cooperative replicated cache scheme. In the following, we present an aspect `CollaborativeCachePolicy` (see Fig. 4.12) realizing cache groups which replicate data among them as introduced in the previous example (see "simple strategy" above), but which also uses summaries to selectively get data from farther caches outside the cache group. These two sets of hosts are represented by groups `cacheGroup` and `summaryHosts`, respectively (line 3). Summary information is shared between hosts of the cache groups and the farther hosts at the border using AWED's group sharing feature. This provides for a concise integration of summaries (no additional code to handle summary management a replication) and is appropriate because summary-based caching algorithms only infrequently update summaries. A simpler (and at times more inefficient) sharing mechanism than for the cached data itself can therefore be used for them.

Overall, the aspect consists of a three step remote sequence `replPolicy` (line 20) which represents the initialization of the cache, the query strategy, and the replication strategy. The sequence first matches the cache initialization, followed by repeated cache accesses and cache replication operations.Concretely, at cache initialization time (see the pointcut at line 8) the two host groups are set up as well as initial summary information (advice at line 25). A

```
1  all aspect CollaborativeCachePolicy {
2
3    group(cacheGroup, summaryHosts) SummaryT summaries;
4    group(cacheGroup, summaryHosts) int cacheMisses = 0;
5    final int THRESHOLD = 1000;
6    ...
7
8    pointcut initCache(Cache c):
9      call(* Cache.init()) && host(localhost) && target(c);
10
11   pointcut getCache(Cache c, String key):
12     call(* Cache.get(String)) && host(cacheGroup)
13        && target(c) && args(key);
14
15   pointcut putCache(Cache c, String key, Object data):
16     call(* Cache.put(String, Object)) && target(c)
17        && args(key, data) && !on(jphost) && on(cacheGroup)
18        && !within(CollaborativeCachePolicy);
19
20   pointcut replPolicy(Cache c):
21     replP: seq(s0: initCache(c) -> s1 || s2
22                s1: getCache(c, k1) -> s2,|| s2
23                s2: putCache(c, k2, val) -> s1 || s2);
24
25   after(Cache c): step(replP, s0) && target(c) {
26     if (c.isDomain(cache)) addGroup(cacheGroup);
27     if (c.isDomain(border)) addGroup(summaryHosts);
28     initializeSummaries(); }
29
30   around(Cache c, String key): step(replP, s1) && args(c, key) {
31     Object obj = c.get(key);
32     if (obj == null) {
33       obj = proceed();
34       if(obj != null) {
35         c.put(key, obj);
36         cacheMisses++; } }
37     return obj; }
38
39   syncex around(Cache c, String key):
40       step(replP, s1) && args(c, key)
41         && on(summaryHosts, awed.combination.and(
42                               awed.targets.filter(summaries),
43                               awed.result.getFirst)) {
44       if (cacheMisses > THRESHOLD)
45          updateSummaries(summaries.getHosts())
46       return c.get(key, o); }
47
48   before(Cache c, String key, Object o):
49       step(replP, s2) && args(c, key, o) {
50     c.put(key, o); }
51 }
```

Figure 4.12: Aspect-based cooperative cache

cache access using `getCache` (line 11) first looks up the value locally, and, if not found in the cache group, attempts to acquire it from the outside. The advice at line 30 first accesses the local cache. If the data is not found, it calls `proceed` to trigger a synchronous remote advice (line 39) which (because of the `on` clause) queries hosts of group `summaryHosts`: this is achieved using a filter selecting hosts whose summaries indicate that the value should be present (the filter is specified as an argument in the `on` pointcut, in order to define a selection policy for the group of hosts declared in the same `on` construct). The remotely executed advice body returns the query result and requests updates of the summaries if a threshold number of cache modifications has been exceeded (this accounts for the basic property of infrequent updates of summary information, see [FCAB00]). The replication within the cache group is achieved as above by matching put operations using the pointcut `putCache` (line 15). This pointcuts triggers the advice at line 48 which executes a put operation on all hosts in the cache group which are different from the host where the original put occurred.

4.4 Advanced examples: JBoss cache extension and refactoring

This section presents results on two evaluations using aspects with explicit distribution for refactoring (i) and extension (ii) of the standard replication strategy of JBoss Cache. These examples show how to cleanly modularize the crosscutting functionalities discussed before and how to support extension of legacy replication strategies.

4.4.1 Refactoring of JBoss replication code

In chapter 3 we have identified replication and transaction as two functionalities contributing to crosscutting within JBoss cache. We show here a re-implementation of the basic replication mechanism, and the transaction-guarded replication mechanism. As described in chapter 3, the basic replication mechanism replicates all changes on the cache. In the other hand, in transaction-guarded replication, the replication protocol first executes the transaction locally (a transaction may be composed of several changes on the cache), requiring, in particular, only the acquisition of local locks. Once the commit method is invoked in the current transaction, all the modifications are replicated to all the nodes in a cache replication group: at remote hosts a prepare phase is executed and, if successful in all nodes, the transaction is committed. If a prepare phase in any node fails, the transaction is rolled back at all nodes.

Basic replication mechanism

Figure 4.13 shows a re-implementation of the basic mechanism of replication proposed by JBossCache. The pointcut definition matches a call to the method `_put` in the class `TreeCache` on all the hosts that are not the joinpoint-host. Once the joinpoint is matched, a replicating advice is executed only if the matched joinpoint was not inside a transaction.

Transaction-guarded replication

Figure 4.14 shows a re-implementation of the transaction-guarded replication strategy of JBoss Cache. AWED allows to cleanly modularize the transaction-guarded replication protocol in a single aspect. The first advice is executed when a local `commit` method is invoked,

```
1  all aspect TreeCacheTransactionReplication {
2      pointcut cachePut(GlobalTransaction gtx, String fqn, Object key, Object value):
3       call(* org.jboss.cache.TreeCache._put(GlobalTransaction , Fqn , Object , Object, boolean))
4           && args(gtx, fqn, key, value, boolean) && !on(jphost)
5           && !within(TreeCacheTransactionReplication);
6
7  before(GlobalTransaction gtx, String fqn, Object key, Object value):
8          cachePut(gtx, fqn, key, value){
9      testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
10     if(gtx==null){
11         System.out.println("Replicating without tx..");
12         cp.getTc().put((Fqn.fromString(fqn), key, value);
13     }
14 }
```

Figure 4.13: Refactoring the JBoss replication code (principle)

the code is used to extract information from the context and invoke an auxiliary method used to trigger the replication protocol. The aspect then defines two pointcuts that represent a `RemoteCommit` and a `RemoteRollBack` respectively. With those pointcuts definitions in place, three synchronous `around` advices are defined to implement the two phase commit protocol. The first `around` advice is used to rollback the transaction in the joinpoint host if an error occurs during the prepare phase. The second `around` advice executes replication handling at other nodes through a remote call to `preparePhase()` and raises an exception in case of errors. The third `around` advice commits the transaction. Note that two different selection classes are used during the protocol execution. The class `AWED.hostselection.AllSuccessfull` is used to execute the synchronous advice in all hosts and returns an exception if any of the hosts reports an exception. The class `AWED.hostselection.All` is used to execute the advice in all hosts. Finally an advice that matches any remote rollback is put in place to assure local rollback of transactions when the prepare phase fails in any of the remote hosts.

4.4.2 Extension of the JBoss replication strategy

As a second evaluation, we realized an extension to the JBoss standard replication strategy: data should only be replicated to nodes that have explicitly requested it, *i.e.*, new objects inserted in a cache group are not replicated spontaneously.

Figure 4.15 shows an aspect definition using AWED that enables lazy replication in a JbossCache. The aspect defines two piontcuts `startSelectiveMode` and `finishSelectiveMode`, respectively used to define the start and end events of the selctive replication mode. The implementation uses a sequence pointcut `replS` which implements a distributed protocol: this protocol starts selective replication mode, followed by local interception of get operations (which explicitly indicate interest in some data). Then the aspect intercepts local replication operations, replicating only when some interest to the data has been registered, until selective mode is terminated. Data which is to be replicated is selected through the advice applied before get operations (step 2 in the sequence `replS`) which registers interest to the data.

```
1  all aspect LocalTransacCommit {
2     pointcut localCommit():
3         call(* org.jboss.cache.transaction.DummyTransaction.commit()) && on(jphost)
4         && !within(org.emn.djasco.cache.TransacReplication);
5
6     before(): localCommit() {
7         testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
8         GlobalTransaction gtx=cp.getTc().getCurrentTransaction();
9         TransactionEntry entry= cp.getTc().getTransactionTable().get(gtx);
10        List modifications= new LinkedList(entry.getModifications());
11        ReplicationHelper.getInstance().txReplication(gtx, modifications);
12    }
13
14    pointcut remoteCommit(GlobalTransaction gtx, List modifications):
15        call(* org.emn.djasco.cache.ReplicationHelper. txReplication(GlobalTransaction, List))
16        && args(gtx, modifications) && !within(org.emn.djasco.cache.TransacReplication);
17
18    pointcut remoteRollBack(GlobalTransaction gtx, List modifications):
19        call(* org.emn.djasco.cache.ReplicationHelper. rollBack(gtx, modifications))
20        && args(gtx, modifications) && !within(org.emn.djasco.cache.TransacReplication);
21
22   //Local advice roolbaks if something fails
23   //in the two phase commit protocol
24    syncex around(GlobalTransaction gtx, List modifications):
25        remoteCommit(gtx, modifications) && on(jphost){
26        testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
27        try{
28            proceed();
29        }catch(Exception e){
30            ReplicationHelper.getInstance().rollBack(gtx, modifications);
31            throw e;
32        }
33   }
34   //Execute the prepare phase in all the hosts
35   syncex around(GlobalTransaction gtx, List modifications):
36        remoteCommit(gtx, modifications) && !on(jphost, AWED.hostselection.AllSucessfull){
37       testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
38       try{
39           ReplicationHelper.getInstance().preparePhase(gtx, modifications);
40       }catch(Exception e){
41           ReplicationHelper.getInstance().rollBack(gtx, modifications);
42       throw e;
43       }
44       proceed();
45   }
46   //commits definetly in all the hosts
47   syncex around(GlobalTransaction gtx, List modifications):
48        remoteCommit(gtx, modifications) && !on(jphost, AWED.hostselection.All){
49       testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
50       try{
51           ReplicationHelper.getInstance().commitPhase(gtx, modifications);
52       }catch(Exception e){
53          //don't do nothing if fails commiting locally
54       }
55       proceed();
56   }
57
58   before(GlobalTransaction gtx, List modifications):
59      remoteRollBack(gtx, modifications) && !on(jphost){
60          //roll back the transaction
61      }
62 }
```

Figure 4.14: Refactoring the JBoss replication code (detailed excerpt)

```
1  all aspect lazyReplication {
2      pointcut cacheGet(String fqn):
3       call(* org.jboss.cache.TreeCache.get(String))
4       && args(fqn) && on(jphost) && !cflow(* lasyReplication.*(*));
5
6      pointcut cacheReplicate(MethodCall method_call):
7       call(* org.jboss.cache.interceptors.ReplicationInterceptor. replicate(MethodCall))
8       && args(method_call) && on(jphost);
9
10     //this pointcut can be matched by any execution on
11     // any host, is not host restricted
12     pointcut startLasyModeEvent():
13      call(* AWED.utils.LazyMode.start());
14
15     //finish lazy mode event
16     pointcut finishLasyModeEvent():
17      call(* AWED.utils.LazyMode.end());
18
19     pointcut replPolicy(String fqn, MethodCall method_call):
20        replS: seq(s1:startLasyModeEvent() -> s4 || s3 || s2 ,
21                   s2: cacheGet(fqn) -> s4 || s3 || s2,
22                   s3: cacheReplicate(method_call) -> s4 || s3 || s2,
23                   s4: finishLasyModeEvent() -> s1)
24
25     around(String fqn):
26        step(replS, s2){
27           IamInterestedIn(fqn);
28           return proceed();
29        }
30
31     around(MethodCall method_call): step(replS, s3){
32         Method meth=method_call.getMethod();
33         if(meth.equals(TreeCache.prepareMethod) ||
34             meth.equals(TreeCache.commitMethod) || meth.equals(TreeCache.rollbackMethod)) {
35             return proceed();
36         }
37         else if(amIInterested(method_call)){
38         return proceed();
39     }
40 }
```

Figure 4.15: Extending the JBoss replication strategy

```
1  //Class DummyTransaction
2  public void commit() throws ... {
3      ...
4      try {
5          boolean outcome=notifyBeforeCompletion();
6          ...
7          notifyAfterCompletion(doCommit? Status.STATUS_COMMITTED :
8                                Status.STATUS_MARKED_ROLLBACK);
9          ....
10     }
11     finally {
12         // Disassociate tx from thread.
13         tm_.setTransaction(null);
14     }
15 }
16
17 //Class SynchronizationHandler
18 public void beforeCompletion() {
19     TransactionEntry entry=tx_table.get(gtx);
20     ...
21     // REPL_SYNC only from now on
22     try {
23         int status=tx.getStatus();
24         switch(status) {
25         ...
26         case Status.STATUS_PREPARING:
27         try {
28             MethodCall prepare_method;
29             prepare_method=new MethodCall(TreeCache.prepareMethod,
30                new Object[]{gtx, modifications,
31                        (Address)cache.getLocalAddress(), Boolean.FALSE});
32             runPreparePhase(gtx, prepare_method,
33                             (Address)cache.getLocalAddress(), modifications, false);
34          }catch(Throwable t) {
35              log.warn("runPreparePhase() failed. Transaction is marked as rolled back", t);
36              tx.setRollbackOnly();
37              throw t;
38         }
39         break;}
40     }catch(Throwable t) {
41         throw new NestedRuntimeException("", t);
42     }
43 }
44
45 //Class ReplicationInterceptor
46 protected void runPreparePhase(GlobalTransaction tx,
47                                MethodCall prepare_method, Address coordinator,
48                                    List modifications, boolean async) throws Exception {
49     List rsps;
50     int num_mods=modifications != null? modifications.size() : 0;
51     // this method will return immediately if we're
52     // the only member (because exclude_self=true)
53     if(log.isTraceEnabled()) log.trace(...);
54     rsps = cache.callRemoteMethods(cache.getMembers(), TreeCache.replicateMethod,
55                                    new Object[]{prepare_method},
56                                    !async, // sync or async call ?
57                                    true, // exclude self
58                                    cache.getSyncReplTimeout());
59     if(!async && rsps != null) checkResponses(rsps);
60        // throws an exception if one of the rsps is an exception
61 }
```

Figure 4.16: JBoss cache transaction replication implementation

4.4.3 Comparison to a JBoss-only solution

To conclude this section, let us compare the two previous examples to the implementation based only on JBoss. Figure 4.16 shows three methods from three different classes that respectively participate in the coordination and replication of the prepare phase, of the two phase commit protocol used, and in the host that is starting the replication of the transaction. First, the `commit` method is called in the class implementing the `Transaction` interface (i.e. `DummyTransaction`) this method uses the before-completion and after-completion idiom: if the `beforeCompletion` method of any of the listeners has failed, the `afterCompletion` method used to roll back the transaction. When a before-completion method is called all the listeners are notified, in particular the listener `SynchronizationHandler`. When `SynchronizationHandler`'s `beforeCompletion` method is called the `runPreparePhase` method of the class `ReplicationInterceptor` is invoked. Using reflection and the `Jgroups` API, the remote calls are distributed to all the participating hosts, waiting for an answer in the case of synchronous distribution. Note that this behavior corresponds only to the code executed in the host from which the transaction originates. Once a remote host receives a message to replicate, the `TreeCache` class invokes the `replicate` method that belongs to the class `ReplicationInterceptor`. In that class, the transaction is decomposed, each call is then replicated using a local call inside a local transaction, using the normal interceptors chain, see chapter 3. This code excerpt shows that replication code gets scattered in multiple classes and tangled with other functionalities as well as infrastructure support code (*e.g.*, code implementing interceptors and listeners).

Contrary to the tangled implementation of transaction and replication code in JBoss Cache, our aspect refactoring clearly separates replication and transactions (transactions appear, apart from their setup, only in exception handlers). Concretely, we have been able to refactor the scattered replication functionality and the corresponding transaction handling which amounts to around 500 LOC in JBoss into one aspect of around 100 LOC. Furthermore, our aspect does not require any particular transaction management but reuses the default transaction management of JBoss Cache. Finally, the second example shows how extensions can be easily integrated using AWED, in particular, because distributed protocols can concisely be expressed using sequence aspects.

4.5 Implementation

Two of the main features the AWED language requires from its underlying middleware implementation are the ability to intercept joinpoints from other hosts and the capability to execute advice on other hosts. A static aspect compiler, as for instance employed by AspectJ [KHH+01], is not well suited to facilitate a flexible distributed AOP platform, as this setup requires all aspects to be present on all the applicable hosts at compile or weave time. As such, all hosts need to be known and fixed in advance, which losses a lot of flexibility. A dynamic AOP approach however, allows to dynamically add/remove hosts and aspects, which is an important feature for large-scale distributed systems.

Therefore, we have chosen the JAsCo [SVJ03] dynamic AOP framework as an implementation platform for the AWED language. JAsCo can be easily extended and provides highly efficient advice execution through its Hotswap and Jutta systems [VS04]. Furthermore, JAsCo already natively supports a model of stateful aspects based on finite state machines [VSCDF05], which can be extended to support distributed sequences as well(JAsCo's

sequences and AWED's sequences are both based in the model for regular aspects presented by Douence, Fradet, and Südhot [DFS02, DFS05]). We start the discussion with an overview of AWED implementation.

4.5.1 AWED implementation overview

AWED is designed to support the implementation of homogeneous distributed applications, but also to support the modification of distribution semantics in heterogeneous legacy distributed applications. Therefore, to allow a flexible platform adaptable to the heterogeneity of the distribution realm, we have used several extensible library/framework during implementation. Concretely, regarding communication design, we have based the internal communications of AWED runtime infrastructure in a flexible protocol stack, provided by JGroups [Ban02], and developed semantic connectors to address the interaction with legacy code in other communication frameworks (*e.g.*, Java's RMI). Figure 4.17 shows a high level view of the main components used during the implementation of AWED. To support remote pointcuts, and remote deployment of aspects we use JGroups [Ban02], an advanced library for group communication, and JavaAssist [Chi00] for bytecode manipulation. The compiler, aspect support, and dynamic weaving are granted by JAsCo's owns infrastructure. Finally, to implement distributed control flow over RMI applications, RMI, and parameter passing we use Java's network and Remote Method Invocation APIs.

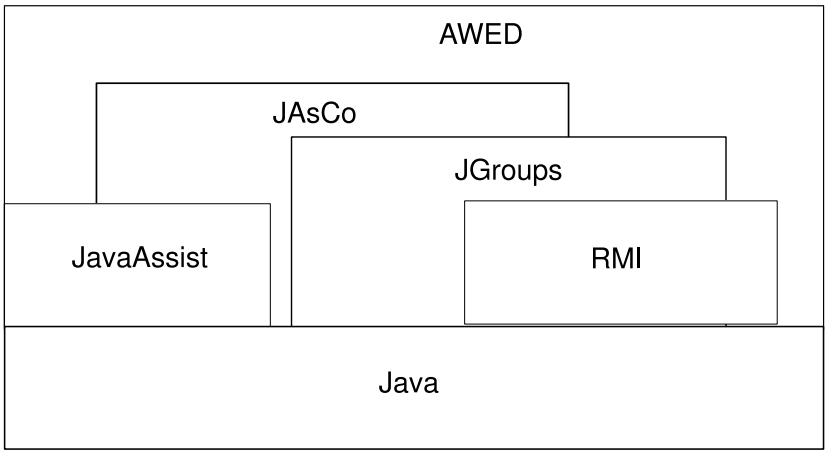


Figure 4.17: Main components used in the implementation of AWED.

Overview of supported features

Table 4.5.1 shows an overview of the main features of AWED and their implementation techniques. The rest o this chapter briefly introduce the AWED run-time architecture, then presents a detailed description of the implementation of main features (see table 4.5.1), and finally discusses some optimizations inherited from JAsCo.

4.5.2 AWED architecture

AWED is a dynamic aspect language that weaves aspects with classes at load time and allows aspect deployment and undeployment at execution time. Its implementation presents

Features	Implementation Techniques
Remote pointcuts	join point controlled advice chain invocation
Control flow	Java's customized sockets
Remote Sequences	JAsCo sequences and remote pointcuts
Remote advice	Based on activation of deployed aspects
Aspect distribution	Aspect dynamic deployment and bytecode propagation
Aspect state sharing	AWED aspects
Parameter passing, remote references	AWED aspects
Causal sequences	Logical clocks, Vector clocks

Table 4.2: Overview of AWED's features and their corresponding implementation techniques

an optimized partially evaluated interpreter for distributed aspects. Figure 4.18 shows the overall architecture, *i.e.*, its compilation chain and the main structures of its runtime framework. In the top left part of the figure we can see how the application and aspect code is compiled into Java bytecode. The bytecode is then read by AWED's instrumentation and transformation framework at load time, producing a version of the application that is instrumented at the necessary joinpoints (here a subset of the method calls). When executing the instrumented application, and once it reaches an instrumented joinpoint, the application dispatches joinpoint notifications to the `Registry` framework that takes care of the recognition of distributed sequence pointcuts (see detailed description bellow). This framework passes the joinpoint notification to each aspect instance, that, in turn, evaluates each joinpoint to match pointcuts and to apply advice. An AWED runtime framework, including a registry, is running at runtime on each logical host,*i.e.*, JVM. In order to support remote pointcuts each host controls the advice chain of join points occurring on that host. As such the compiled code at load time will be coordinate remote registries to execute remote advice using an extension of the JGroups framework [Ban02]. This part of the infrastructure contains all necessary support for remote regular sequence pointcuts. In figure 4.18, we have also detailed the communication framework (see the box labelled "JGroups" in the figure). In the figure we show a traditional protocol stack that supports different protocols, including the User Datagram Protocol (UDP). This architecture also allows the composition of different aspects over a single join point, in particular AWED uses precedence declaration inherited from JAsCo to address this particular issue. Thus, in a single JAsCo controller the instantiation behavior, the concrete pointcuts, and the precedence of several aspects can be declared.

Detailed runtime behavior of the registry framework

The run-time architecture can be distributed using two different strategies: either a single connector registry is kept for all hosts or each host separately maintains a dedicated connector registry. The first solution has the advantage that a single registry is responsible for the aspect execution, whereas the second solution requires the distributed connector registries to be synchronized. In general, a central entity is considered to be a problematic solution in a distributed setting, as it inherently does not scale and can become a performance bottleneck. Therefore we choose to deploy a separate connector registry at each host (see figure 4.19).

Every connector registry is responsible for the locally intercepted joinpoints and its locally deployed aspects. In order to allow aspect execution on remote joinpoints, the intercepted

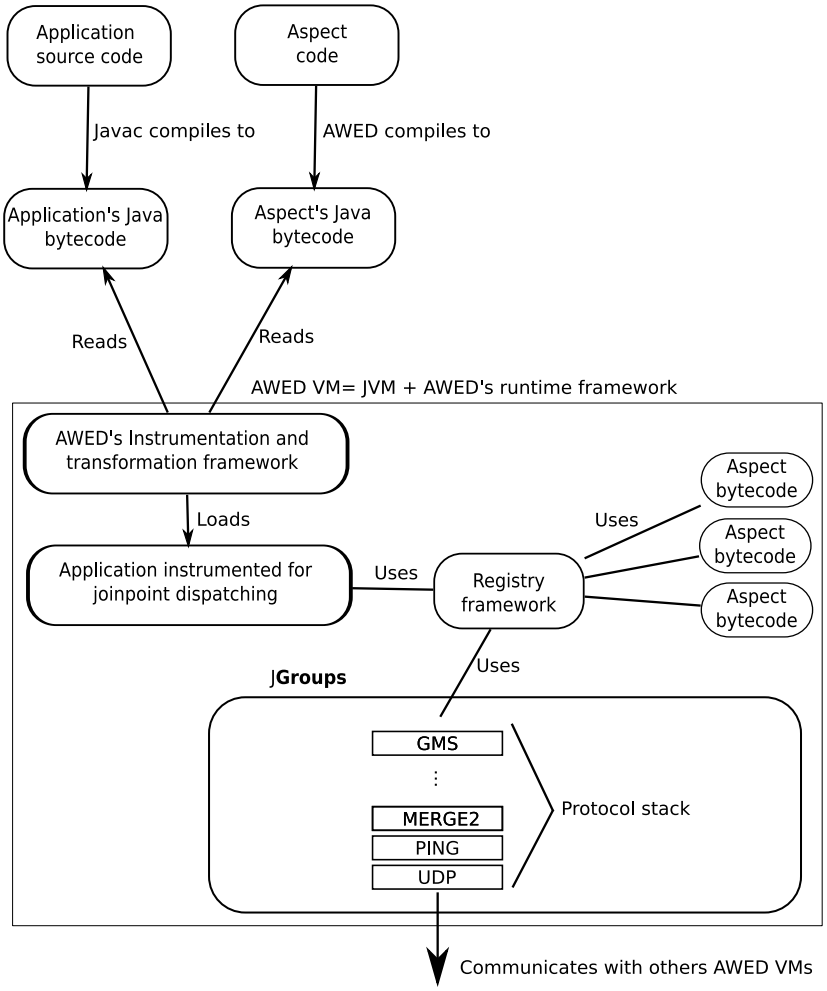


Figure 4.18: AWED architecture.

joinpoints need to be sent to the other hosts. Likewise, in order to allow aspect execution on remote hosts, the aspects need to be distributed as well.

4.5.3 Remote pointcuts

In order to execute advice that trigger on remote joinpoints, the joinpoint information should be distributed to all interested hosts. To this end, a plugin for the connector registry has been implemented that: 1) intercepts all joinpoints, 2) prepares them for transmission and 3) sends them to the remote hosts. Joinpoints need to be prepared before transmission as not all joinpoint information might be transmittable. Our current system uses Java Serialization to transmit objects from one host to another. For this, all contextual information (e.g. callee, actual arguments) that is neither serializable nor primitive or that has been labeled to be passed by value is transformed according to the remote reference model (see Sec. 4.5.3). In a last step, the joinpoint information is sent to the remote hosts. In order to locate and send this information to other interested hosts, the JGroups framework is employed [Ban02].

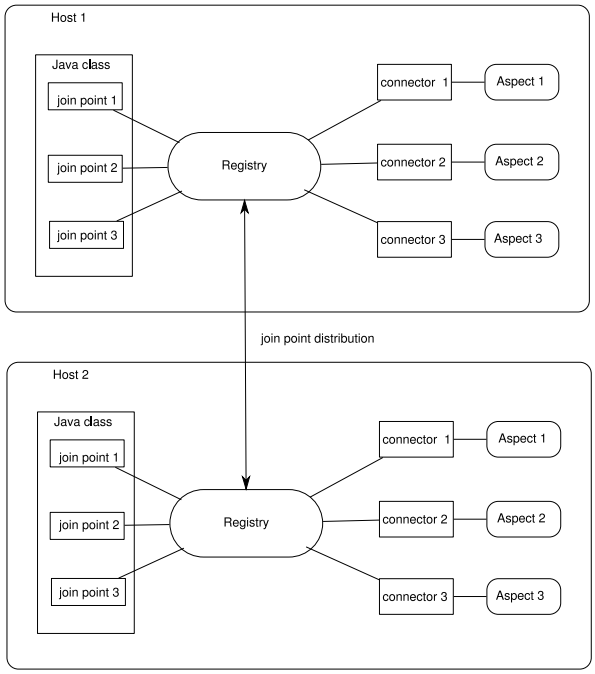


Figure 4.19: Detailed runtime behavior and architecture of the registry framework.

Remote References

The AWED model offers object transparency, and allows objects to be passed by ref as well as by value, this transparency includes the reflective information of joint points. Java's standard infrastructure for remote method invocation RMI is not sufficient for our purpose, because it does not provide object transparency and does not support different parameter passing modes. Furthermore, when using RMI's remote referencing model, the advice implementation cannot access advice variables nor reflectively query joinpoint information that is not serializable. Although this is an important limitation, it is typical in such distributed environments. For instance, arguments of Java RMI method invocations need to be serializable or primitive as well. Therefore, we have decided to implement the AWED infrastructure on top of RMI, extending it with aspects to support object transparency, *i.e.*, local and remote objects are treated in the same way.

RMI requires objects to be explicitly instrumented in order to make them behave as remote objects. To avoid this complication AWED uses two indirection layers on top and below RMI. The layer on top of RMI provides remote delegation proxies that redirect remote method calls to actual methods in the objects that are referenced remotely. These proxies are deployed in hosts where the actual objects reside. The layer below RMI provides redirection-proxy aspects that are deployed in the hosts where the remote references are actually used.

As mentioned above the layer on top of RMI is implemented using indirection proxy objects (see Figure 4.20). An indirection proxy is a remote object, remote in the RMI sense, that holds a reference (local reference) to a POJO (referenced) object. This proxy exposes a remote method that passes the calls to the referenced object using the reflection API. Using this technique, no previous instrumentation is needed to reference an object remotely. Any object can be selected to be passed by reference in a remote invocation. However, this

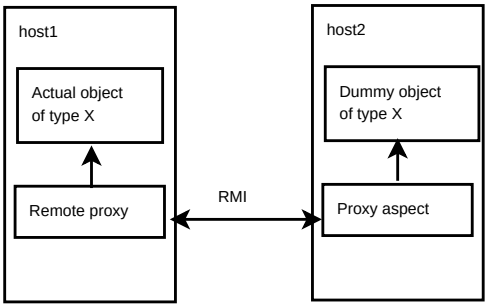


Figure 4.20: AWED's remote reference model implementation. The actual object is referenced by a remote proxy object and represented by a dummy object of the same type in the remote host. Such a dummy object is instrumented by an aspect that redirects calls to the remote proxy. The remote proxy then redirects calls to the actual object.

technique poses a problem: The RMI remote reference is now of type remote indirection proxy and not of the type of the referenced object.

This problem of type mismatch is solved using a proxy aspect. We have modified the reflective joinpoint model to include the type information of the remote reference of each argument in each particular joinpoint. Using the type information, dummy objects are instantiated to behave as the remote references. These dummy objects are instrumented using a dynamic proxy aspect that holds a reference to the actual remote proxy. Any call that is made to the dummy object is caught by the aspect and redirected to the remote proxy. This proxy receives the call in the remote host and redirects it to the actual object that is being referenced remotely. Note that the dummy object and the actual object that is being referenced remotely are of the same type. Figure 4.20 illustrates this mechanisms.

Sequences

JAsCo's stateful aspects, *i.e.*, finite-state based sequences [VSCDF05], have been extended to a distributed setting in order to implement AWED's distributed sequences. This is a rather straightforward process, because the state of a sequence pointcut is not managed by the JAsCo run-time infrastructure (deployed locally, see figure 4.19), but by the aspect itself. The aspect intercepts remote joinpoints, matching its stateful pointcut description in a similar way as for joinpoints matching regular pointcuts. Afterwards, the internal state is updated by firing the relevant transition(s) in the internal state machine.

4.5.4 Aspect Distribution

In order to execute advice on remote hosts, the aspects themselves should also be distributed to the host(s) in question. One solution would be to force an administrator to manually deploy the aspects on every applicable host. However, as an advice execution host sometimes depends on complex expressions with several variables, it might be difficult to manually deploy the aspects onto remote hosts in an optimal fashion. For instance, deploying aspects to hosts where they can never be applicable is useless and wastes the system resources of those particular hosts. Hence, the DJAsCo extension automatically distributes the aspects to

all remote hosts that might be applicable. (The current implementation still uses automatic aspect deployment on all hosts, however, like the advice chain is controlled by the registry where the join point was generated, the extension of current distributed loader into a distributed lazy class loader is unproblematic.) When a new host joins, the DJAsCo run-time infrastructure detects this event, and the host automatically receives the possibly applicable aspects. Likewise, when new aspects are deployed at a particular host, they are automatically deployed at the relevant remote hosts. It is possible to avoid this automatic deployment of aspects on remote hosts by marking them with the `single` modifier.

Technically, JGroups is again employed to transfer the aspects and to be informed of changes in the network setup such as newly joined hosts. In contrast to joinpoints, aspects are class-based entities and it suffices to send the class byte-code to the remote hosts. Hence, possible serialization problems are avoided.

4.5.5 Asynchronous Advice and Futures

When advices execute asynchronously, the return value might not yet be available while the advised base application expects a value. Consider e.g. a `compPrice` method that computes the price of a certain order where a discount around advice that substracts a fixed value is applied. The invoker of `compPrice` expects a computed price, but in case the around advice executes asynchonously, this value might not yet be available. A common technique to solve the return value issue with asynchronous invocations is the use of futures [RHH85]. A future object represents the return value and waits for its availablity in case the value is explicitly claimed. The main problem in our case is that the base application does not expect a future. The caller of `compPrice` for instance expects a numeric value. Therefore, we employ transparent proxies that wrap the future object. The transparent proxy is an instance of the same type as the expected return value. An AWED aspect is applied onto the transparant proxy and intercepts all method executions. For the first invocation, the future is claimed and thus synchronized with the remote advice. Afterwards, the aspect simply redirects the invocation to the available result of the future. Using the technique, the base application remains oblivious of the existance of the future object. The performance impact is limited to the equivalent of one additional method invocation. It would be possible to improve this even more by employing the technique proposed by Pratikakis et al [PSH04].

4.5.6 State Sharing

The AWED language supports state sharing between different instances of the same aspect type regardless of the location and/or VM where they are executed. In order to implement `local` sharing, we generates one master field on every host for each locally shared aspect field. All aspect instances of the aspect type on that host automatically refer to that field using Java RMI. Field queries and updates are automatically redirected to the shared field. This redirection takes place by employing another AWED aspect that is dynamically generated and weaved when an aspect, defining a shared field, is being deployed. Visibility modifiers for the fields (such as `private`) do not hinder the sharing implementation because they can be overridden at run-time.

The `global` state sharing could be implemented in a similar fashion, *i.e.*, having one globally shared field. However, this solution suffers from a serious robustness problem as all aspect instances of the same type would rely on one specific host that holds the shared field.

Therefore, the local master fields are explicitly synchronized using yet another AWED aspect that is automatically generated at deployment time. Figure 4.21 illustrates a simplified version [6] of this global state sharing aspect. The after advice is triggered for every state change of the `myfieldname` field of the `myaspectname` aspect. The advice is executed on every host except on the one that triggered the joinpoint. As such, the state change is propagated to all other hosts. The advice implementation first fetches an aspect instance of the given type on the host where it is executing and than changes the value to the newly assigned value. Because all aspect instances on that host refer to the same field, the new value is immediately propagated to all aspect instances of the `myaspectname` type on the host at hand.

```
1 all aspect StateSharing {
2   pointcut stateChanged(Object value):
3    set(myaspectname.myfieldname) && args(value) && !on(jphost);
4
5   after(Object value): stateChanged(value) {
6       myaspectname aspectinstance = myaspectname.apectOf();
7       aspectinstance.myfieldname=value; }
8 }
```

Figure 4.21: State sharing as a AWED aspect

4.5.7 Optimizations

In addition to the connector registry, JAsCo's run-time architecture consists of two other systems: HotSwap and Jutta. HotSwap allows to dynamically install traps only at those joinpoint shadows that are subject to aspect application. When a new aspect is deployed, the applicable joinpoints shadows are hot-swapped at run-time with their trapped equivalents. Likewise, the original byte code is reinstalled when an aspect is removed and no other aspects are applicable on the joinpoint shadow at hand. Jutta on the other hand, is a just-in-time compiler for aspects that allows to generate a highly optimal code fragment for every joinpoint shadow. By caching these code fragments, an important performance gain is realized. Both of these systems are inherited from JAsCo. The JAsCo run-time weaver, based on HotSwap and Jutta, is able to compete performance-wise with statically compiled aspect languages such as AspectJ, while still preserving dynamic AOP features [FVSB05]. A major concern of the AWED design is to preserve compatibility of the weaver with these two tools, in particular to enable the optimization of remote pointcuts.

The JAsCo HotSwap and Jutta systems are compatible with the AWED architecture. Because all aspects are present at every applicable host (even aspects that might execute their advice elsewhere), the local HotSwap system still knows where to insert traps. Aspects that do not define pointcuts relevant for the local host are not deployed and are of no interest to the HotSwap system as they do not induce newly trapped joinpoints. Remote joinpoints are represented similarly to local joinpoints. Hence, the Jutta system is still able to generate and cache a code fragment for executing the joinpoint locally. As such, apart from the network delay and serialization/deserialization cost, no additional overhead is required for remote pointcuts and distributed advice executions.

[6] Notice that this aspect has been simplified for presentation purposes. For example, it does not cope with the fact that aspect types might not be present on every host.

Chapter 5

Invasive Patterns

5.1 Introduction

Software patterns have proven a versatile tool for program development, be it for the development of application designs [GHJV94], architecture descriptions [TS+03] or program implementations [E+06]. Design patterns have been very successful in the domain of sequential, and in particular object-oriented applications. Similarly, pattern-based development methods have been extensively applied in the parallel computing domain for the derivation and implementation of massively parallel algorithms [TS+03, SDGS96, Col89]. However, pattern-based approaches have been much less successful in the domain of distributed programming. In particular, if such patterns are defined over irregular communication topologies and subject to heterogeneous synchronization constraints. Consequently, patterns for distributed programming (see, for instance, patterns for distributed enterprise information systems and grid applications [SSRB00, Cor, E+06]) are often expressed as mere programming recipes. These recipes are not backed up by concrete architecture or implementation entities that can be reused as building blocks for applications.

In this chapter we investigate a major reason for the difficulty in applying programming patterns, which embody common computation and communication patterns, to distributed applications: frequently, applications of such patterns in realistic contexts depend on information on the execution state that is not directly available when the pattern is to be applied. This is, for instance, the case in two frequent cases: (i) in legacy contexts where patterns could be used to improve the application structure but in which instructions for communication instructions and manipulation of related execution state are frequently scattered over numerous places and (ii) in distributed applications that have been designed using less flexible abstractions than provided by communication and computation patterns.

In this chapter we introduce *invasive patterns* for distributed programming. Such patterns essentially provide well-known regular computation and communication patterns but extend them by a built-in abstraction for access to non-local execution state whose access is required to enable pattern applications. We provide evidence that techniques from Aspect-Oriented Programming (AOP) [ACEF04] can be used to augment patterns by structured access to such non-local state.

The chapter is structured as follows. First, we discuss different pattern-based approaches for parallelism and distribution (section 5.2.1). Then, we present a detailed motivation for invasive patterns and corresponding aspect-oriented support based on a detailed analysis of

the use of implicit patterns architectures in the JBoss Cache strategy for replication in the context of transactions, thus extending the analysis presented in chapter 3 (Section 5.3). Section 5.5 introduces a pattern language that allows to concisely define invasive variants of well-known patterns for distributed applications. Section 5.7 briefly sketches a prototype implementation of invasive patterns using a transformation into the AWED language. Finally, we give an evaluation of our approach by discussing how invasive patterns can be used to improve the structure of JBoss Cache and applying them in the context of a benchmark application for grid programming (section 5.7).

5.2 Pattern-based approaches for distributed development

5.2.1 Massively parallel patterns

In 1989 Cole introduced algorithmic skeletons for parallel computations [Col89]. These skeletons described general high level algorithms based on problem decomposition and distribution into parallel processors. These abstract algorithms are instantiated using higher order functions, *i.e.*, functions accepting functions as parameters and as return values. Thus, a programmer could select a skeleton as the main program and provide specific domain functions, following skeleton specification, to create a program which reused the parallel algorithm described by the skeleton. An example of Cole's work is the Fixed Degree Divide & Conquer skeleton (FDDC). This skeleton was inspired by the common solution of dividing complex problems in simpler sub problems. For example, having a list, the problem of ordering such list can be solved using a recursive algorithm: first divide the list in two sublists, then apply the ordering algorithm to those sublists recursively, *i.e.*, dividing the sublists in smaller sublists, and finally merge the two resulting lists in order. A general algorithmic skeleton for this kind of problems is defined as follows:

$$\begin{aligned} FDCC\ indivisible\ split_k\ join_k\ f &= F \\ \text{where } F\ P &= f\ P, \text{ if } indivisible\ \text{P} \\ &= join_k\ (map\ F\ (split_k P)), \text{ otherwise} \end{aligned}$$

Thus, for any problem domain where the problem is of type *prob* and the solutions are of type *sol* the programmer must provide the following functions:

$$\begin{aligned} indivisible &: prob \rightarrow boolean \\ f &: prob \rightarrow sol \\ split_k &: prob \rightarrow [prob] \\ join_k &: [sol] \rightarrow sol \end{aligned}$$

Where *indivisible* is a function which inspects the problem and decides whether it can be solved recursively. The function f is the base case (*i.e.*, indivisible case). $split_k$ is the function which decompose a problem always in k subproblems, easier to solve. $join_k$ is the function knowing how to join the k solution of the subproblems, and k is an integer fixed by the programmer and respected by functions $join_k$ and $split_k$. This integer is used to give some rigidity and homogeneity to the distribution schema, facilitating the mapping from the decomposition three into a system of parallel processors. For example, figure 5.1 shows graphically a representation of a binary tree distribution in a grid of processors. This distribution of process represents a symmetric distribution of processes for the simplest FDDC algorithm (k

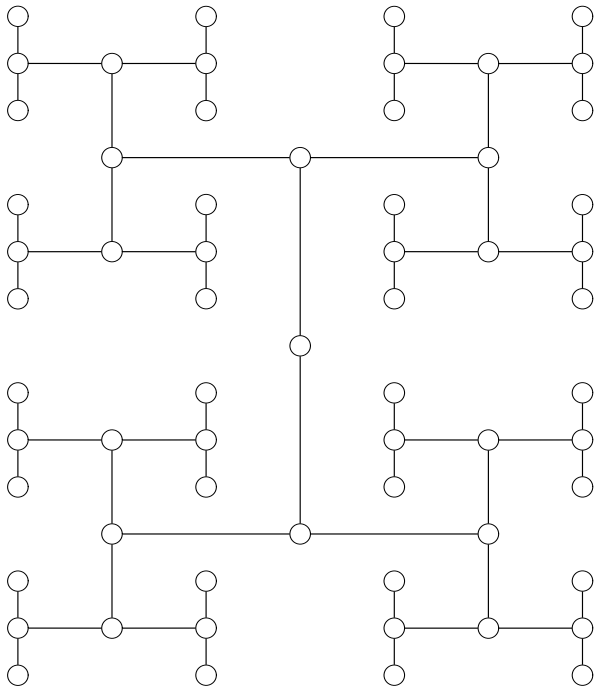


Figure 5.1: H tree topology

= 2).

Note that this approach essentially relies on an underlying regular communication topology and uses a homogeneous synchronization model, two properties that do not hold for the applications we are targeting. Furthermore, crosscutting access to execution state on which pattern application depend are not addressed explicitly. Recent work in parallel patterns have not focussed on these restrictions. Instead, it has focused on the application of such pattern-based parallelism to large-scale imperative applications (see, *e.g.*, [TS+03, SDGS96]). In particular, in the design of supporting tools for the automatic generation of code.

Our approach extends these approaches by addressing the problems of homogeneity and regular communication models. Concretely, we provide dynamic mechanisms for the definition of distributed algorithms instantiated concretely in an aspect based language.

5.2.2 Architectural patterns for distributed applications

Patterns have been instantiated in terms of different artifacts and at different levels of abstractions. One of these instantiations is the conceptual documentation (in catalogues) of recurrent practices to solve particular problems. In the domain of distributed software several pattern catalogues have been proposed [BMR+96, SSRB00, AMC+03]. These catalogs describe either architectural patterns, designs patterns, idioms, or all of them.

Architectural patterns describe the decomposition of a system into subsystems(*e.g.*, [BMR+96, SSRB00]) providing guides to organize its relationships, and specifying their responsibilities. For example, the *pipes and filter* architectural pattern from [BMR+96], provides a structure for systems processing a stream of data. In such case each filter receives an stream of data as an input, and outputs a modified stream of data (see figure 5.2). That stream can then be used as the input of the following filter. Each filter is connected to the next filter using pipes.

Pipes serve to synchronize and communicate output between filters. Thus, several systems can be constructed using different sequences of filter connected by pipes. A popular instantiation of this pattern can be found in Unix-like systems where for example the command `ls | less` is connect by a pipe (`|`), and composed by `less` and `ls` subsystems. The resulting processing pipeline will take as input the directory contents and will output them into the terminal allowing forward and backward navigation.

This pattern allows concurrent processing of the stream. Thus, the next filter in the chain can start processing the stream before the previous filter finishes the process of all the stream. Figure 5.2 shows a graphical representation of such pattern. In the picture the stream flows from left to right passing through all the filters.

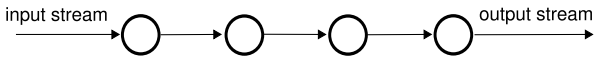


Figure 5.2: Pipes and filter pattern

Architecture description languages

Distributed applications are often built using rich middleware structures which provide basic services for the implementation of typical computation and communication patterns. However, the middleware infrastructures do not include dedicated support for pattern definitions. In the domain of grid computing, for instance, Globus one of the most popular middleware for grid architectures, uses the resource specification language RSL [Glo] to support the deployment of applications. The corresponding specifications include, among others, information related to the application (location of the executables, number of instances of a program, etc.), as well as information on the execution environment (names of computers, job submission methods, working directory, environment variables, etc.). However, in contrast to the notion of invasive patterns advocated here, computation and communication patterns have to be programmed in an ad hoc manner. In particular because RSL cannot describe connection constraints between the various parts of an application that depend on execution state that is encapsulated by the distributed nodes.

A common means to overcome such restrictions are architecture description languages, in which patterns or pattern-like structures are used for system specification and implementation. This approach is particularly widespread in component-based systems. Such systems are typically constructed from a set of components that are interconnected through well-defined ports. To mention just one of the many corresponding approaches, a Corba Component Model (CCM) [CCM] architecture description, for instance, contains one or more components as well as two specific pieces of deployment-relevant information: component placement information and component interconnection information. Furthermore, CCM and other component-based models contain mechanism that are used to implement patterns, *e.g.*, mechanisms for the implementation of asynchronous broadcast services. However, as for grid middleware, these programming abstractions are not made explicit in the architectural description that defines the interconnection properties and, in contrast to our approach, no explicit means for the embedding of pattern-like interconnection structures in crosscutting contexts is provided.

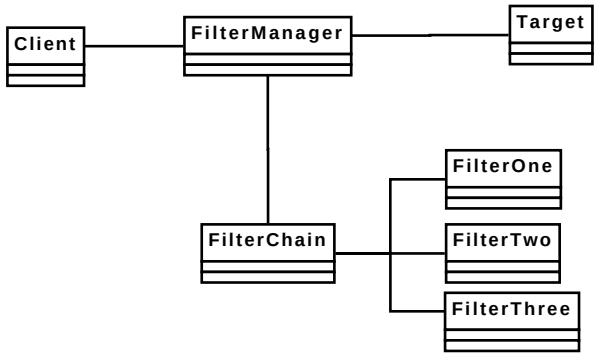


Figure 5.3: Standard startegy of filter pattern [AMC+03]

5.2.3 Design Patterns for distributed applications

Design patterns in their most popular instantiation propose object oriented descriptions of recurrent practices in object oriented programming [GHJV94]. Such patterns are particularly widespread in component-based systems, *e.g.*, the CORBA and J2EE platforms [CCM, AMC+03]. These component systems provide communication and concurrency mechanisms that are used to implement patterns, *e.g.*, for the implementation of asynchronous broadcast services. However, most of these descriptions provide a local structural organization of code, assuming implicit communication support by the underlying framework (*e.g.*, J2EE).

Figure 5.3 shows the class diagram of the filter pattern, as presented in *J2EE Core patterns book* [AMC+03]. The pattern is conceptually similar to the Filter and Pipes pattern presented before. However, in this case we found concrete instances represented as classes. The figure shows a class diagram with a class `FilterManager` that creates filter chains (classes `FilterChain` and `Filter`), and processes the requests between the client and the target object. Even though, the pattern does not explicitly define the client as remote, the context of the book (J2EE patterns) and the detailed description imply a remote interaction (*e.g.*, an interaction between an http client and an application server).

Patterns can have several implementation strategies. For example, figures 5.4 and 5.5 show two different sequence diagrams for two different implementation strategies. Figure 5.4 shows a sequence diagram that matches exactly the class description presented in figure 5.3. The `Client` sends a request to the `FilterManager`, which creates a `FilterChain`, which applies each filter over the request. Finally, the `FilterManager` forwards the filtered request to the target object. In the other hand, figure 5.5 shows an implementation strategy similar to that described in chapter 3 for JBoss Cache. This strategy uses a Decorator pattern [GHJV94] to create a chain of filters where each filter wraps the next one, calling directly its `Execute` method. At the end of the chain the target object receives the filtered request.

5.2.4 Aspect oriented pattern approaches

Design and architectural patterns propose abstract reusable solutions for software applications and systems architecture. However, design and architectural patterns are not instantiated in reusable software artifacts, instead, they serve as documented guides. Furthermore, during the implementation of such patterns several issues have to be considered. We are particularly interested in traceability and reusability. Traceability of design pattern is often lost in the

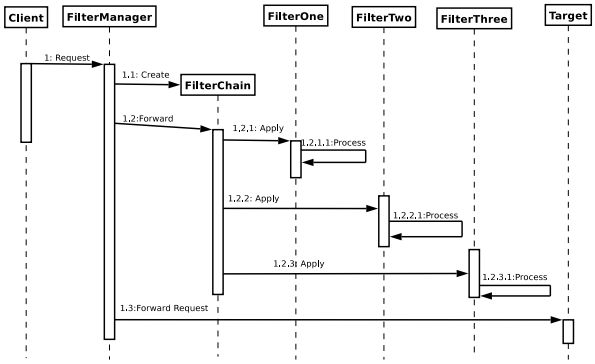


Figure 5.4: Sequence diagram of standard startegy of filter pattern [AMC+03]

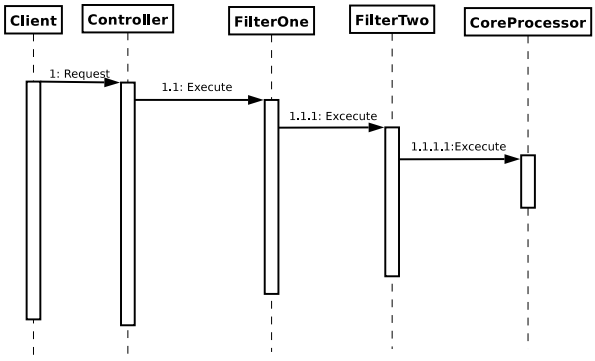


Figure 5.5: Sequence diagram for custom filter pattern: implementation using Decorator pattern [GHJV94]

implementation. As shown in chapter 3, patterns are not easily identifiable at the code level and they do not encapsulate cleanly the concerns they were designed for (similar problems were first identified by Soukup [Sou95] and Bosh [Bos98]).

Regarding reusability, design patterns and architectural patterns are provided as design guides. These design guides are not, in general, first class elements of the language. Hence, they can not be reused in other implementations. This problem was identified early in research literature and addressed with specific language constructs and extensions (*e.g.*, see [Sou95, Bos98]). Another solution to address reusable implementation of design patterns was proposed by Hannemann and Kiczales in [HK02]. In particular, they show that several quality attributes, such as locality of definition and code reusability, of GoF pattern implementations can be improved through usage of AspectJ.

A number of approaches have been put forward that use distributed AOP for the modularization of crosscutting functionalities in distributed and concurrent applications (see chapter 2). These approaches, while in principle being able to express invasive patterns as we propose, can only do so by modularizing crosscutting functionalities using separate aspects for each node in a distributed system. Our approach, through its pattern language is much more declarative by directly expressing distribution-relevant relationships within single aspects, thus resulting in more concise programs that facilitate program understanding and maintenance.

5.3 Pattern-like structures in distributed middleware

In this section we present modularization problems of pattern-like computations in JBoss Cache.

5.3.1 Pattern-like structures in JBoss Cache

We have analyzed the occurrences of pattern-like computation structures and the dependencies of such pattern-like structures on the underlying execution state in JBoss Cache, as an extension to the study presented in chapter 3. In the following we briefly describe the results of our analysis of software patterns that are used implicitly in this infrastructure.

Explicit design using design patterns

As presented in chapter 3, JBoss Cache implementation consists of two main parts: (i) a main class `CacheImpl` that represents the main data structure, a tree with a hash table on each leaf, and (ii) a set of filters that is used to implement the major part of the behavior of non-functional requirements, mainly transactions and data replication. Each call to the `CacheImpl` API is first transformed into a method call object using a reflection mechanism. Once this object is created, it is passed to a chain of filters where each filter adds some behavior, *e.g.*, optimistic locking is added by the transaction filter. Eventually, the filtered method call is performed (see chapter 3 for more details). This simple design is implemented using a filter pattern. The components of such a pattern can be found explicitly instantiated in the class structure of JBoss Cache. However, the comunication patterns and their relations are not explicitly found in the code.

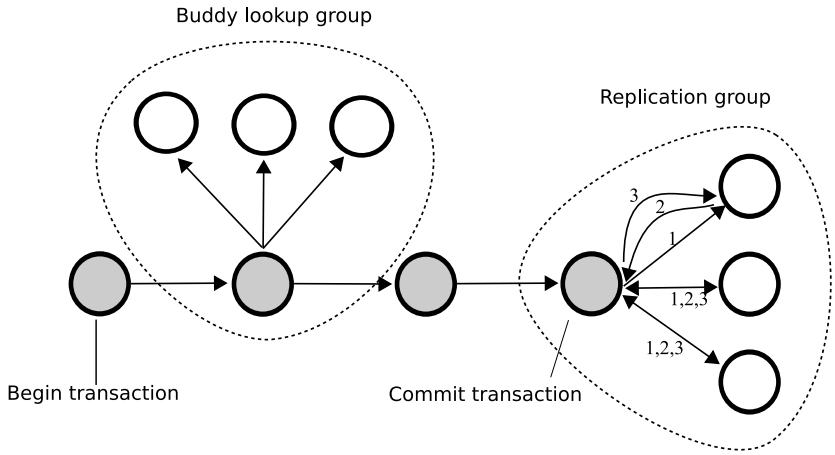


Figure 5.6: Architecture of transaction handling with replication in JBoss Cache

Implicit comunication patterns

The current production version (2.0.0) of JBoss Cache has an architecture that can be expressed nicely in terms of patterns using, *e.g.*, a pipeline pattern for transaction control and a farm pattern for replication actions. Note that JBoss Cache provides several configurable feaures, *e.g.*, transaction locking strategies (*e.g.*, pesimistic, optimistic), buddy replication (group of preferred hosts for initial replication), and replication protocol (one phase commit, two phase commit). Here and in the rest of this chapter we assume the cache to be configured for transactions with pessimistic locking and a two phase commit protocol.

Figure 5.6 presents a high-level pattern-based view of the corresponding (similar to architectural patterns) system structure of JBoss Cache. In the figure, a transaction is triggered by a specific method call represented by the first node in the pattern. Then successive calls to `get`, `remove` or `put` methods on the cache are executed and the information is stored for further replication. When a particular value is not present in the cache, the cache looks for the value in a group of selected neighboring nodes, its so-called buddies, illustrated by the three edges starting in the second node of the figure. Once the end of a transaction is reached, the originating cache engages in a two phase commit protocol [LS76, Gra78]. In such a protocol the originating cache sends a prepare message with the transaction control information (edges numbered 1 in the right part of the figure), followed by answers from all buddies confirming agreement or non-agreement (edges numbered 2). Finally, the originating cache sends a final commit or a rollback message depending on the answers it received (edges numbered 3). Note that in this interaction we can identify, at least, two well separated groups of hosts, one for the search of values at buddy nodes and the other for the replication behavior from a node to other nodes.

In chapter 3 we have analyzed the complexity of the implementation of the JBoss Cache framework. In particular, we have shown that replication and transaction instructions are widely scattered over the code base and tangled with one another in numerous places. Even though JBoss Cache conceptually is characterized by a pattern-based structure as shown in Fig. 5.6, the current implementation does not allow conventional communication patterns for distributed systems to be applied due to the scattering and tangling of code. A detailed qualitative analysis of such code leads to the identification of three basic problems:

1. Transactional and replication behavior depends on the state that is stored in different classes. Such a state is modified in scattered pieces code that *e.g.*, reify the current transactional state as mentioned above so that it can later be tested in another class in order to decide which replication action to perform.

2. The relationships governing the interplay between the main concerns, transactions and replication, are not made explicit anywhere in the code. Instead, scattered pieces of code implicitly coordinate these concerns, thus generating tangled code and breaking the modularization aimed at by the JBoss Cache filter mechanisms.

3. JBoss cache includes several distribution-related concerns (*e.g.*, replication, cache loaders and buddy lookup) that require communication between different groups of hosts. Groups overlapping and interactions between different groups generate additional tangling.

5.3.2 Source code representation of pattern-like structures

Figure 5.7 shows a piece of code of the main filter method `invoke` of the `DataGravitationInterceptor` class that is responsible for the so-called data gravitation concern, *i.e.*, buddy lookup. This method clearly exhibits the problems stated above, providing evidence of tangling of three concerns: replication, transactions and buddy lookup. The code uses a common idiom in the JBoss Cache to address transactions control inside a `switch` instruction (lines 7 to 19). The right branch in the switch statement is taken depending on static information in the execution state, *e.g.*, a configuration-time choice between optimistic and pessimistic locking, and dynamic information about the execution state, *e.g.*, the dynamic type of the current processed method call. There, in order to calculate the method id (see line 5) the application relies on an ad-hoc mapping that is defined in the class `MethodDeclarations`. Similarly, the choice between optimistic and pessimistic locking is made at configuration time inside the `TreeCache` class as well as part of the class `InterceptorChainFactory` (this choice in turn affects at runtime the configuration of the dynamically created chain of filters).

Note that the corresponding piece of code is found inside the filter class `DataGravitation` and uses data that is calculated in many different places, thus expliciting problem 1 above. The kind of idiom involving `switch` statements (that clearly represent a mismatch between the conceptual pattern-based architecture and its concrete implementation) is scattered over multiple places in the implementation. We have found 67 places where such a switch action is used, and more than 27 places where it occurs in the context of transactions and replication operations (thus providing testimony for the problems 1 and 2 introduced above).

Furthermore, the `DataGravitation` class plays an unexpected role in the two phase commit protocol. A method of type `commitMethod` is processed in order to send a commit message on those caches that are not part of the current buddy group, see line 34 in the `docommit` method (*i.e.*, being subject to problem 3 above). Remember that the `DataGravitation` class was supposed not to control the transactional behavior or the replication of transactions which should normally be performed by the transactions and replication filters.

5.4 Motivation and requirements for Invasive patterns

There is a very large choice of potential basic architectural patterns for distributed programming. A set of basic patterns could be derived, for instance, from schema for the implemen-

```
1  //----- Piece of code in the invoke method of
2  //----- DataGravitationClass
3  try
4  {
5   switch (m.getMethodId())
6    {
7      case MethodDeclarations.prepareMethod_id:
8      case MethodDeclarations.optimisticPrepareMethod_id:
9           Object o = super.invoke(ctx);
10          doPrepare(ctx.getGlobalTransaction());
11          return o;
12     case MethodDeclarations.rollbackMethod_id:
13          transactionMods.remove(ctx.getGlobalTransaction());
14          return super.invoke(ctx);
15     case MethodDeclarations.commitMethod_id:
16          doCommit(ctx.getGlobalTransaction());
17          transactionMods.remove(ctx.getGlobalTransaction());
18          return super.invoke(ctx);
19   }
20 }
21 catch (Throwable throwable)
22 {
23  transactionMods.remove(ctx.getGlobalTransaction());
24  throw throwable;
25 }
26
27 //---- The docommit method in DataGravitation class
28 private void doCommit(GlobalTransaction gtx) throws Throwable
29    {
30       if (transactionMods.containsKey(gtx))
31       {
32          if (log.isTraceEnabled())
33               log.trace("Broadcasting commit for gtx " + gtx);
34          replicateCall(getMembersOutsideBuddyGroup(),
35             MethodCallFactory.create(
36                  MethodDeclarations.commitMethod,
37                  new Object[]{gtx}),
38                  syncCommunications);
39       }
40       else
41       {
42          if (log.isTraceEnabled())
43              log.trace(
44               "Nothing to broadcast in commit phase for gtx " + gtx);
45       }
46    }
```

Figure 5.7: Tangled code of a two phase commit (2PC) protocol inside the invoke method of the `DataGravitationInterceptor` class.

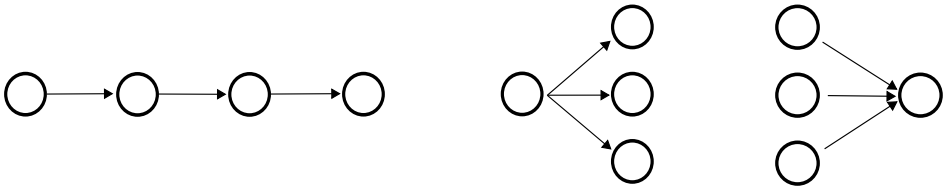


Figure 5.8: Basic patterns

tation of publish-subscribe relationships [EFGK03], from the large pool of patterns that have been studied for distributed and parallel programming [Col89], or from more recent work on patterns for current distributed systems(*e.g.*, grid-based systems [E+06]). However, dependencies as those motivated before for JBoss Cache between transaction-related actions and replication operations cannot simply be modularized using standard patterns for workflow-related computations, such as pipelining, farming out or gathering computations as illustrated in Fig. 5.8. In the figure circles denote calculations that possibly take place on different hosts and edges denote communication. In fact, taking scattering and tangling of transactions and replication into account does not fit the common interpretation of such patterns in which each circle denotes a well-defined entity, in our motivating example such entity is some nicely modularized piece of code within JBoss Cache.

In such cases effective support for a pattern-based programming style should allow:

- The definition of patterns to include access to the data it depends on but that is defined at other places in the underlying distributed program.
- Allow such patterns to be applied possibly at numerous places in a program, including remote places.
- Distributed work and data flow (see distribution and parallelism in chapter 2)
- Coordination. First, coordination of the different parts inside an Invasive pattern, *e.g.*, data access. Second, external coordination between invasive patterns. (see concurrency in chapter 2.)
- Allow composition of multiple patterns.

5.5 Invasive patterns

5.5.1 Structure and design

We pursue the idea of invasive pattern at the programming level by extending skeleton based patterns, as presented by Cole [Col89] (see section 5.2.1), with a notion of aspects to modularize crosscutting access. Concretely, we provide a notion of reusable architectural-communication patterns (*e.g.*, pipe, farm, gather) and the corresponding programming artifacts. The resulting notion of invasive patterns is illustrated in Fig. 5.9 for the case of a `gather` pattern. On the three nodes on the left hand side, different invasive-access (represented by dashed lines) are used to access information that is then prepared by "source" computation (represented by the filled rectangles) to be sent to the right hand side node. Once all relevant

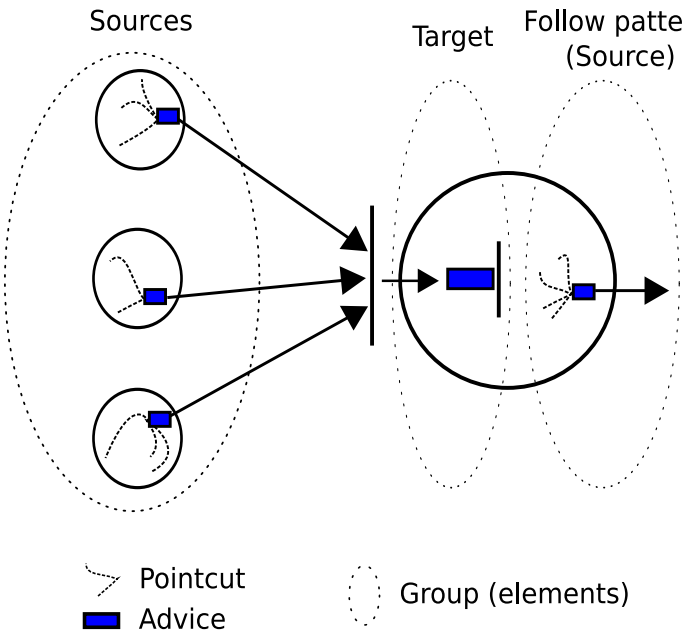


Figure 5.9: Invasive patterns

data has been passed to the right hand side node, a "target" computation is used to integrate the transmitted data with an existing or new computation on the target node. In order to support the declarative definition of such crosscutting access, we leverage results on so-called stateful pointcut languages [DFS02] that enable matching of sequences of execution events to be defined using expressive languages, in particular finite-state automata.

Besides a definition of basic invasive patterns a suitable notion of pattern composition is needed. Reconsider the (abstract) architecture of transaction handling with replication in JBoss Cache (see Fig. 5.6) this architecture can naturally be expressed in terms of compositions of the three basic patterns introduced above, where the steps denoted 1–3 in the figure correspond, for instance, to two applications of the `farm` pattern and one application of the `gather` pattern. Our approach supports the compositional construction of such architectures from the basic patterns on the programming and the implementation level. As discussed in Sec. 5.3.1, this architecture is essentially hidden in the actual JBoss implementation. Our approach can therefore be seen as a means to make explicit such architectures, and thus help program understanding and maintainability.

5.5.2 Synchronization

A crucial issue concerning invasive patterns as motivated before it is shown how the different activities (invasive-access, local and remote computation) are synchronized with one another. In this section, we first discuss corresponding design choices and then present our language for the definition of invasive architectural patterns.

The definition of distributed algorithms using patterns over a state-based programming paradigm essentially depends of the correct synchronization on the different parts of invasive patterns and between different invasive patterns. Pattern-based computations can be synchronized roughly at three different levels:

1. *Synchronization within an invasive pattern.* Most basically, a target computation is executed only after a rendez-vous synchronization of all source computations. In the case of the `gather`-pattern shown in Fig. 5.9, the target computation is started only after the three target hosts have "agreed" to trigger it. Second, a target computation may be executed in a synchronous or asynchronous fashion. Synchronous execution of parts of the `pipe` pattern of Fig. 5.8a corresponds to a fully sequential (a.k.a. batch) computation, while its asynchronous execution corresponds to a pipelined computation. We support both behaviors.

2. Computations involving *consecutive executions of patterns* may be synchronized with one another. The `gather` pattern may, for instance, be synchronized with the execution of the following pattern. The following pattern is represented in the right hand side node by the invasive-access (the dotted lines), the source computation (the small rectangle) and the arrow leaving the node to the right. Execution of a follow pattern on a node n must start after control of the previous pattern has entered n (otherwise the two pattern executions could not be said to be consecutive) but may be reasonably started either when the target computation of the previous pattern is started or when it terminates.

3. Most generally, synchronization constraints may be imposed on *arbitrary segments of pattern compositions*. Such general constraints are useful, *e.g.*, because computations may be executed on the same host and therefore give rise to problems, such as race conditions. Such synchronization strategies cannot, however, be defined simply in terms of individual patterns as considered here. In this work we present a sequential pattern constructor, however, other pattern constructs considering different synchronization specifications are subject of future work, *e.g.*, a construct that receives two patterns as parameters, and starts the two patterns in parallel depending of specific state conditions.

Summarizing, we provide in this chapter explicit support for intra-pattern synchronization and synchronization between consecutive pattern executions. We do not, however, provide general synchronization strategies over pattern compositions because they are difficult to comprehend and may easily lead to performance bottlenecks or even deadlocks. We envision that specific properties over pattern compositions can be analyzed and enforced in terms of the more restricted means for synchronization we introduce here.This issue is, however, considered as future work, interested readers revised preliminary work on the formalization of invasive patterns [BNDNS08].

5.6 Pattern language

Because crosscutting of non-local execution state that enables pattern applications is at the heart of invasive patterns, Aspect-Oriented Programming [Kic96, ACEF04] seems a promising approach for the modularization of such patterns and the corresponding data access. Thus, we have decided to provide programming artifacts in form of an aspect based programming language. At the core of a such language we have a kind of parameterized higher order function, similar to those provide by Cole [Col89], that receives aspects, patterns, and groups of hosts as parameters. Such functions define the composition of pattern sequences that structure the common communication patterns defined above and their composition.

$$
\begin{array}{lcl}
P & ::= & \texttt{patternSeq}\ G_1\ A_1\ G_2\ A_2\ \ldots\ G_n \\
G & ::= & H\ G \ \mid\ P\ G \ \mid\ \epsilon \\
A & ::= & \texttt{aspect}\ \{\ \texttt{around}((\mathrm{Id}^*)\text{: } PCD \quad SourceAdvice \quad [\texttt{sync}]\ TargetAdvice\ \} \\
PCD & ::= & \texttt{call}(MSig) \ \mid\ \texttt{target}(Id) \ \mid\ \texttt{args}(Id+) \\
 & \mid & PCD\ \&\&\ PCD \ \mid\ PCD\ ||\ PCD \ \mid\ !PCD \\
 & \mid & Seq
\end{array}
$$

Figure 5.10: Pattern language

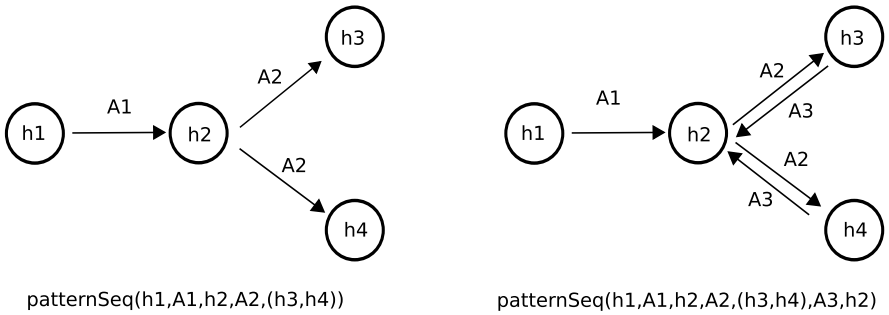


Figure 5.11: Pattern Compositions

5.6.1 Syntax and informal semantics

We are now ready to introduce the pattern language we have designed that realizes the above design choices. Figure 5.10 shows the syntax of our pattern language.

Patterns

The pattern constructor `patternSeq` takes as argument a list $G_1\ A_1\ G_2\ A_2\ \ldots\ G_n$ of alternating group and aspect definitions. Each triple $G_i\ A_i\ G_{i+1}$ in this list corresponds to a pattern application that uses the aspect A_i to trigger the pattern in a source group G_i and realize effects in the set of target hosts G_{i+1}. A group G is either defined as a set of host identifiers H or through a pattern constructor term itself. In the latter case, the group is defined as the source or target group of the constructor term depending on the argument position the term is used in. This constructor enables the definition of the basic patterns shown in Fig. 5.8: `pipe` as a `patternSeq` from a single host to another, `farm` as a `patternSeq` from a single host to several hosts and `gather` as a `patternSeq` from several hosts to a single one. Pattern compositions can be defined with more complex `patternSeq` terms. For instance, the left hand side of Figure 5.11 defines a composition `pipe` then `farm`, and its right hand side defines a composition `pipe`, `farm` then `gather`. These examples make clear it is easy to define sophisticated compositions akin to the architecture of transaction handling in JBoss Cache (cf. Fig. 5.6).

Aspects

Aspects A that define the behavior of invasive patterns specify a pointcut PCD that allows the modularization of crosscutting code that triggers a pattern, and defines a source advice and a target advice executed respectively on the source and target groups of a pattern. Advice can

```
1 aspect ReplicatingCache {
2     around(String fqn, String key, String value):
3           call(* Cache.put(..)) && args(fqn, key, value)
4         { proceed(); }
5         { (NamingCache.lookup("registered Cache")).put(fqn, key, value)}
6    }
```

Figure 5.12: A Session Profiling Aspect

be parametrized by bound values (see `args` below). An advice is a standard block of code, but a source advice can call the matched base call with the `proceed` keyword. Otherwise, the base call triggers the aspect but the execution of the corresponding base method is skipped. When a `sync` annotation is used to qualify target advice, the base program execution on source hosts is not resumed before the end of the target advice. The default behavior is an asynchronous execution.

Pointcuts

We consider pointcut definitions that are essentially restricted to matching of method call joinpoints. This pointcuts may, additionally, extract target objects with `target`, extract arguments of calls with `args`, and use logical compositions of pointcuts. Following the paradigm of stateful pointcuts [DFS02, BNSV+06a] (and unlike AspectJ [asp08, KH+01]) pointcuts may match sequences (non-terminal *Seq*) of calls in the base program execution. We omit the syntax of sequences for now, but they are basically defined in terms of a finite-state automaton by declaring its states and by labelling state transitions with pointcuts (see AWED syntax for detailed definition). Note that remote pointcuts are not allowed, and the matching has local semantics.

Let us consider a small example. The aspect `ReplicatingCache` in Figure 5.12 is used to replicate calls to the `put` method over an object of type `Cache`. When the method `put` is called the local advice performs it (through a call to `proceed()`) and the target advice replicates the call to the `put` method. This aspect can be applied using the expression:

patternSeq(h1, ReplicatingCache, h2, ReplicatingCache, (h3, h4))

The left part of figure 5.11 shows a graphical representation of such a composition (for this example in the figure $A1 = A2 = ReplicatingCache$). The resulting composition defines a two level cache that first replicates calls to the `put` method from host `h1` to host `h2`, and then from host `h2` to hosts `h3` and `h4`. This two level caching is a common architecture enabled through complicated configuration time deployments in JBoss Cache.

5.7 Implementation

In order to implement the pattern language presented in the previous section, support for three main mechanisms is necessary: (i) aspects providing a modular abstraction for invasive access on the source hosts and triggering activities on target hosts, (ii) flexible means for synchronization within individual patterns and between consecutive pattern executions, and (iii) the concise definition of the communication topologies of patterns.

We have implemented invasive patterns using AWED, which provides direct support for most of the necessary features and allows the accommodation of the remaining ones based on its native abstractions.

Intuitively, a farm pattern can be mapped to an AWED aspect using a pointcut expression as

`call`(* *.put()) && `host`("sources") && `on`("targets"),

there, the `call` pointcut matches calls to the *put* method. The pointcut `host`("sources") matches the join points (events) that appear in a host that belongs to the *sources* group. Finally the pointcut `on`("targets") triggers the execution of the advice in hosts that belong to the *targets* group. AWED also supports the `Seq` pointcut that allows the specification of specify finite-state automata that permit to match sequences of join points in distributed applications. The sequence constructor is used to map direct uses of *Seq* pointcuts of our pattern language (see Fig. 5.10) and to implement *rendez-vous* synchronization in gather-like patterns. The *Seq* construct allows the definition of pointcut triggered automatons. Regarding the rendez-vous implementation, the *Seq* construct is used to implement an automaton matching any particular ordering of messages coming form a set of hosts.

5.7.1 Transformation of invasive patterns into AWED

We have developed a formally-defined transformation from our aspect language into executable AWED programs. Figure 5.13 formally defines the central part of the transformation of the pattern language defined in Fig. 5.10 into AWED aspects. Even though AWED does not have a formally defined semantics, the transformation into AWED provides informal semantics for the pattern language.

The transformation mainly consists of rules for the two main concepts of our pattern language: pattern sequences `patternSeq` in the pattern language (see Fig. 5.10) as well as aspects that govern the behavior of patterns on source and target hosts. A remark on the use of fonts in the definition: underlined terminals belong to the transformation language, while not underlined terminals belong either to the pattern language (if they appear on the left hand side of a defining equation) or to the AWED language (if they appear on the right hand side).

Transforming the `patternSeq` construct

A `patternSeq` term (cf. def. $\mathcal{T}_P$) results in the transformation of each individual pattern definition ($A_0\ G\ A\ G_2$) where A_0 denotes the aspect defining the pattern that precedes and triggers the current one and A denotes the behavior-defining aspect of the current pattern that is applied to the source group G and the target group G_2. In order to cope with nested patterns (remember G can contain hosts but also `patternSeq` terms), functions $\mathit{first}, \mathit{last}$ respectively extract the source and target hosts to which the aspect A is applied.

The function $\mathcal{T}_A[\![]\!]$ transforms the aspect that defines the pattern behavior. The resulting AWED aspect consists of two pointcuts for the source hosts (*spc*) and target hosts (*tpc*), two corresponding advice definitions ($\mathit{sad}, \mathit{tad}$), and an empty function `triggerNext` that is used to trigger the execution of the following pattern by the current one. This transformation implements invasive patterns as follows:

$\mathcal{T}_P[\![\texttt{patternSeq}\ gas]\!] = \underline{\texttt{foreach}}\ (A_0\ G\ A\ G_2)\ \underline{\texttt{in}}\ gas\ \underline{\texttt{do}}\quad \mathcal{T}_A[\![A]\!](last(G), first(G_2))(A_0)$

$\mathcal{T}_A[\![\texttt{aspect}\ \{\ (\texttt{around}(hps)\ pcd\ sad\ syncMode\ tad)\ \}]\!](G_1,\ G_2)(precAsp) =$

```
all aspect createName() {
   pointcut spc(T_AL[[hps]]):                                  // source pointcut
      Seq(precAsp.triggerNext(); T_PCD[[pcd]]);
   pointcut tpc(T_AL[[hps]]):                                  // target pointcut
      PermutationSeq(G1,T_PCD[[pcd]]);
   around(hps): spc(T_AL[[hps]]) && host(localhost) { // source advice
      sad;
   }
   syncMode after(hps): tpc(T_AL[[hps]]) && on(G2) { // target advice
      triggerNext();
      tad;
   }
   void triggerNext() {}
}
```

$\mathcal{T}_{PCD}[\![\texttt{call}(M)]\!] = \texttt{call}(M)$

... *// other pointcut constructors also valid in AWED*

$\mathcal{T}_{AL}[\![as]\!]$ = *// argument list translation*

// Auxiliary functions

createName	*// creates a fresh name*
first	*// returns group of source hosts of a pattern expression*
last	*// returns group of target hosts of a pattern expression*
PermutationSeq	*// seq. constructor matching all*
	// permutations of a set of joinpoints (part of the AWED library)

Figure 5.13: Transformation into AWED

- The source pointcut *spc* first waits for the triggering event `triggerNext` of the previous pattern and then matches for the pointcut *pcd* given as part of the pattern definition.
- The target pointcut *tpc* uses the pointcut constructor $PermutationSeq(G, p)$ from the AWED library that matches any permutation of events matched on hosts on the group G. This implements a rendez-vous.
- The source advice executes the advice body on the source hosts (which are identified by `localhost` after deployment). The argument list of the corresponding advice in the pattern definition, a list of (host, argument list)-pairs, has to be transformed to a suitable Java representation ($\mathcal{T}_{AL}[\![]\!]$, whose straightforward definition is not shown).
- The target advice first triggers execution of the following pattern and then executes the body *tad* specified in the pattern definition. The argument list of this advice must also be transformed as for the source advice.

The formal definition in terms of the transformation above has served two main purposes. First, it provides a quite *simple but precise definition of our current implementation* of invasive patterns. Second, it allows a precise characterization of *two fundamental synchronization properties* of invasive patterns that have been informally introduced in the previous work:

- *Target advice is only executed after all source activities have terminated.* This property is ensured by use of the $PermutationSeq(G, p)$ pointcut constructor. This pointcut only matches after events of all source hosts have been matched and therefore implements the rendez-vous of all source hosts before the execution of the remote advice on a target host.
- *Two consecutive pattern applications in a pattern sequence should be serialized.* This is ensured by triggering pattern executions by preceding patterns. The above transformation interprets this synchronization constraint by triggering the source aspect of the following pattern as soon as the target advice of the preceding pattern is executed, which is useful, in particular, if follow aspects can react on events introduced through patterns. It is, however, also be possible to trigger the execution of the follow pattern after termination of the target aspect of the preceding pattern.

These two properties, together with the explicit representation of the topology of pattern compositions, pave the way for the investigation of synchronization properties of complex pattern compositions, such as wave-like progress of pattern compositions. This is, however, the subject of future work.

5.8 Evaluation

In this section we evaluate our approach by presenting how invasive patterns can be used to restructure transaction handling and replication in JBoss Cache. We first show how to implement these concerns using the proposed pattern language, thus making explicit their pattern-based structure. We then briefly discuss the resulting implementation in AWED. Third, we qualitatively evaluate the resulting pattern-based implementation by discussion the difference in conciseness of the original and new implementation. Finally, we briefly discuss first results of benchmarking we have performed by executing the refactored implementation of JBoss Cache using the current AWED implementation [AWE08].

```
1   gCaches = {H1, H2, H3}
2   pipe([h],
3        Atransac,
4        farm(
5          gather(
6            farm([h], Aprepare, sync gCaches-[h]),
7            Apresp,
8            [h]),
9          Acommit,
10         gCaches-[h])
11       );
```

Figure 5.14: Pattern-based definition of the JBoss Cache two phase commit

5.8.1 JBoss Cache revisited

Invasive patterns allow the concise expression of the essentials of the pattern-based architecture for transaction handling and replication in JBoss Cache as shown in Fig. 5.6. Concretely, we have implemented support for transactions with pessimistic locking and the two phase commit protocol using invasive patterns.

The corresponding solution is formulated in terms of a nested composition involving four pattern expressions, see Fig. 5.14. First, we apply a `pipe` pattern to be able to relate the start of transactions with the replication operations, *i.e.*, the start node and the final replication group, respectively, of Fig. 5.6. Once a commit is encountered, a `farm` pattern is used to farm-out the prepare phase of the two phase commit protocol. Then, a `gather` pattern is used to collect the answers from the involved buddy caches. Finally, after all answers have been received we use again a `farm` pattern to distribute the final decision of commit or rollback. The code in the figure defines this algorithm for three replicated caches. Note that replication can be triggered from any of the three caches. Once the triggering node (`h` in the algorithm) is selected the expression `gcaches-h` represents the group of caches without the triggering one.

Invasive access required to make this solution work are provided by the involved aspects. As a typical example, consider the aspect `Atransac`, shown in Fig 5.15, that explicitly relates the start of transactions with the replication code. It defines, in particular, the pointcut `transaction` that is used to capture sequences of transaction and replication-relevant events. The pointcut defines an automaton that first waits for a call to `begin` of a transaction. Next, the automaton waits for a `put`, `remove`, `get` or `end` actions that interact with the cache. If the end state is reached, the sequence is restarted in order to handle a new transaction.

Figure 5.16 shows the pattern-defining aspect `Aprepare` that farms out the prepare information of the two phase commit protocol. Occurrences of calls to the `prepare` method are matched and executed (because of the call to `proceed` in the source advice). On the target hosts, the target advice executes the prepare phase followed by the invocation of an agreement or disagreement method, depending on the answer of the target caches. The aspect takes care of transactions that perform nested calls in the prepare method using the `cflow` pointcut construct: this constructs forbids new replication actions within the dynamic extent of an open call to the `prepare` method.

To complete the implementation in the pattern language, we describe the aspects that provide the gathering and the final farming-out of decisions. Figure 5.17 shows the `Aresp`

```
1  all aspect Atransac perthread {
2
3    String transacId;
4    DataStorage transacData;
5    DataStorage currentAction;
6
7    private DataStorage storeAction(String s, Object fqn, Object key, Object value ) {
8      if(transacData == null) {
9      transacData = new DataStorage(s, fqn, key, value);
10     currentAction = transacData;
11     } else {
12       currentAction = currentAction.setNext(new DataStorage(s, fqn,key, value));
13     }
14     return currentAction;
15   }
16
17   pointcut transaction(): seq(
18     sbegin: call(* Transaction.begin(..))
19             && host(localhost) > sput || sremove || sget || sEnd;
20     sput: call(* TreeCache.put(..))
21             && host(localhost) > sput || sremove || sget || sEnd;
22     sremove: call(* TreeCache.remove(..))
23             && host(localhost) > sput || sremove || sget || sEnd;
24     sget: call(* TreeCache.get(..))
25             && host(localhost) > sput || sremove || sget || sEnd;
26     sEnd: call(* TreeCache.rcommit(..))
27             && host(localhost) > sbegin;
28   }
29
30   before sbegin () { /** Do Noting */ }
31
32   around sput () {
33     System.out.println("asp inside put");
34     Object[] args= thisJoinPoint.getArgumentsArray();
35     storeAction("put", args[0], args[1], args[2]);
36     return proceed();
37   }
38
39   around sremove () { /* Similar to sput */ }
40   around sget () { /* get actions are not stored */ }
41
42   around sEnd () {
43     transacId = Thread.currentThread().getName();
44     PrepareHelper ph = new PrepareHelper();
45     ph.send(transacData, transacId);
46     return proceed();
47   }
48 }
```

Figure 5.15: Pointcut definition for transactional behavior in the pipe aspect

```
1  aspect Aprepare {
2    org.jboss.cache.TreeCache tc = CacheRegistry.getInstance().getCache();
3
4    around(DataStorage d, String txId):
5      call(* PrepareHelper.send(..)) && args(d,s) &&
6      !cflow(call(TransactionManager.prepare(..)))
7
8      // Source advice
9      { proceed(); }
10
11     // Target advice
12     { TransactionManager tm = TransactionManager.getInstance();
13       PrepareHelper ph = new PrepareHelper();
14       try{
15           tm.prepare(d, txId, tc);
16         ph.respAgree(txId);
17       } catch(Exception e) {
18         ph.respNotAgree(txId);
19       }
20     }
21 }
```

Figure 5.16: 2PC invasive aspect `Aprepare`

aspect that is in charge of gathering the responses to the prepare phase. The advice now receives a list of pairs consisting of, for each pair, a host and the parameters of each matched join point (each cache can answer `respAgree` or `respNotAgree`). The example shows pseudo code for the remote advice where all the answers are considered: if any of the answers was a `notAgree` a rollback is triggered. In any other case, a final commit is triggered. Finally the `Acommit` aspect is a trivial version of a farm aspect that publishes the final commit or rollback. Note that the aspect does not take into account the result of commit or rollback phases, as this is also the current behavior in JBoss Cache.

These solutions represent a fully decoupled message oriented solution. Each cache receives, consumes and sends messages without blocking or waiting for remote answers. The synchronization and distribution requirements are handled automatically by the pattern language.

Implementation using AWED. The result of the transformation[1] of the pattern program shown in Fig. 5.14 is a set of AWED aspects that implement the replication under pessimistic locking. Each aspect of the pattern-based solution is translated into an AWED aspect that modularizes source and target parts of a pattern expression. Figure 5.18 presents the resulting implementation of the `Aprepare` pattern-level aspect. In this case the generated source pointcut uses a sequence to explicitly relate the relevant transaction-related event to the call `send` that initiates replication, *i.e.*, farming out the prepare action. The target advice executes the prepare method in the target caches and calls an `respAgree` or `respNotAgree` method to yield the answer. Note that the permutation sequence that establishes the rendez-vous between source and target hosts corresponds in this case just to one possible value.

The implementation of a gathering aspect is different and more complex. Figure 5.19 shows

[1] we have applied the transformation manually for this evaluation but its automation is straight forward.

```
1  aspect Aresp{
2
3     around((h1, Object txId1),
4               (h2, Object txId2)):
5                       (call(* PrepareHelper.resp*Agree(..))
6                  && args(txId);
7     //Sources
8     {
9        proceed();
10    }
11
12    //Targets
13    {
14        if(//any answer was respNotAgree)
15          TransactionManager.getInstance().notAgree(txid);
16        else
17          TransactionManager.getInstance().agree(txid);
18    }
19 }
20
21 aspect Acommit{
22
23    around(Object txId): call(* TransactionManager.*gree(..)) &&
24                args(txId);
25
26    //Sources
27    {
28       proceed();
29    }
30
31    //Targets
32    {
33        if(//Use reflection == notAgreeFinal)
34          TransactionManager.getInstance().notAgreeFinal(txid);
35        else
36          TransactionManager.getInstance().agreeFinal(txid);
37    }
38 }
```

Figure 5.17: 2PC aspects to complete the protocol

```
1  all aspect Aprepare_AWED {
2    org.jboss.cache.TreeCache tc = CacheRegistry.getInstance().getCache();
3
4    Group[] targetGs = {new Group("h1"), new Group("h2"), new Group("h3")};
5
6    pointcut sourcePrepareCall(TransactionData d, String txId):
7          seq(init:call(* Atransac.triggerNext()),
8          pcd: call(* PrepareHelper.send(..)));
9
10   pointcut targetPrepareCall(Transaction tx)(TransactionData d, String txId):
11               call(* PrepareHelper.send(..));
12
13   // source advice
14   around(TransactionData d, String txId): sourcePrepareCall(d, txId) && host(localhost) {
15     proceed();
16   }
17
18   // target advice
19   after(TransactionData d, String txId): targetPrepareCall(tx) && on(targetGs) {
20     TransactionManager tm = TransactionManager.getInstance();
21     PrepareHelper ph = new PrepareHelper();
22     try{
23       tm.prepare(Tx.getTransacData(), Tx.getId(), tc);
24       ph.respAgree(txId);
25     } catch(Exception e) { ph.respNotAgree(txId); }
26     void triggerNext() {};
27 }
```

Figure 5.18: 2PC invasive AWED aspect for the creation of the transactional behavior

```
1  all aspect Aresp perbinding{
2
3   Group[] sourceGs = {new Group("h1"), new Group("h2"),
4             new Group("h3")};
5   boolean allEntries = false, adviceEnd = false;
6   pointcut respPointcut (Object txId):
7                       (call(* PrepareHelper.resp*Agree(..))
8                    && args(txId));
9   pointcut rrespPointcut(Object txId):
10   PermutationSeq(respPointcut(txId),
11                targetGs.diference(
12              GroupHelper.getInstance().getTargetGroup()),
13           GroupHelper.getInstance().getTargetGroup());
14
15  around(Object txId): lpid (Object txId) && host(localhost) {
16    proceed();
17  }
18
19  after(Object txId): rpid (Object txId) && on("h1"){
20     allEntries = true;
21     if(//use reflection if any respNotAgree)
22         TransactionManager.getInstance().notAgree(txid);
23     else
24         TransactionManager.getInstance().agree(txid);
25     adviceEnd = true;
26     void triggerNext(){};
27  }
28 }
```

Figure 5.19: 2PC AWED aspects for gathering the response

the `Aresp` aspect after transformation. The source pointcut matches any call to `respAgree` or `respNotAgree`. The advice attached to this pointcut just proceeds to the target one. The target pointcut is the resulting permutation of the source matches at all hosts belonging to the difference of group `sourceGs` and the target host. The target advice launches the actions to the farm the final commit or the final rollback depending of the gathered answers. This aspect is instantiated using AWED's `perbinding` keyword: thus, there will be an instance for each `txId` value. This way we can handle multiple transactions that JBoss Cache handles using multiple threads.

Multithreading and multiple transactions A particular interesting point is how to handle multithreading and multiple transactions. We handle this using aspect instantiations that help us to limit the scope of matched joinpoints and control the instantiation mechanism of new patterns. In particular the `TransactionAsp` aspect uses `perthread` instantiation mechanism, guaranteeing that it will be one instance per thread and that it will match local actions only in the thread. This is a common implementation technique used in transactions , e.g. Jboss cache implementation, consisting in relating the thread to a transaction. One of the problems of the fully decoupled version of the two phase commit protocol is that there is no control flow relation between messages (opposed to current JbossCache implementation) so we use the `perbinding` mechanism to relate different events with an specific instance of an aspect. Note that the control-flow version of the algorithm can be implemented using

AWED, but we wanted to show how composition and interaction can be easily achieved between different patterns.

5.8.2 Qualitative and quantitative evaluation

In Section 5.3 we have motivated that the current implementation of JBoss Cache is subject to problems concerning modularization, in particular, scattered and tangled code for the control of the transaction and replication concerns. Our solution improves the implementation in all those respects. First, each crosscutting concern is now modeled as an aspect and the choreography and interaction is defined without crosscutting by means of the pattern language (and AWED aspects on the implementation level). Second, distribution issues, coordination and composition of patterns are easily identifiable and modifiable in our solution. These advantages appear clearly in the `Aprepare` aspect: the source pointcut clearly defines the exact context (the sequence of method calls matched in the source pointcut) required to trigger the replication; furthermore, the related actions relevant to replication on different hosts are modularized in the aspect. Overall, our solution facilitates understanding and is easier to extend.

We have measured how our refactored version of JBoss Cache compares quantitatively to the plain JBoss Cache solution. For the corresponding experiments, we have considered transactions with pessimistic locking in JBoss cache. In the original code, there are more than 2674 LOC in 17 classes related to this concern. In our solution, the code consists of 532 LOC in 11 well-modularized aspects and classes: roughly a reduction of 80% of complexity (in terms of LOC). Most of this reduction is due to the fact that the transaction and communication protocol that is scattered and duplicated in `switch` structures is now re-factorized in well modularized entities.

5.9 Grids: a case of study for invasive patterns

In recent years, grids have become a powerful system architecture that allows to execute large-scale applications as diverse as scientific applications or large-scale information systems. This kind of architecture, composed of multiple local federations, provides a highly heterogeneous environment to users [Fos01]. To overcome this heterogeneity issue, grid architectures and applications are typically built using special purpose middleware that allows to bridge between existing, often component-based, infrastructures.

Currently, the development of grid applications using such middleware is frequently hampered by two issues: limited means to describe topologies and lack of support for the invasive composition of legacy components. For instance, grid topologies that underlie grid applications are mostly defined only implicitly through message passing as part of a grid application or using low-level means for topology definition, such as graph constructors whose links to the grid application have to be defined once and for all. As to the composition of legacy grid components, it often requires significant rewriting of the involved legacy components, for example, because the composition requires data to be passed that is not exported by the legacy components.

To overcome these problems in this section we evaluate the applicability of invasive patterns. First, we show how invasive patterns and their aspect-oriented features for explicitly distributed programming can be used to modularize crosscutting accesses in the context of

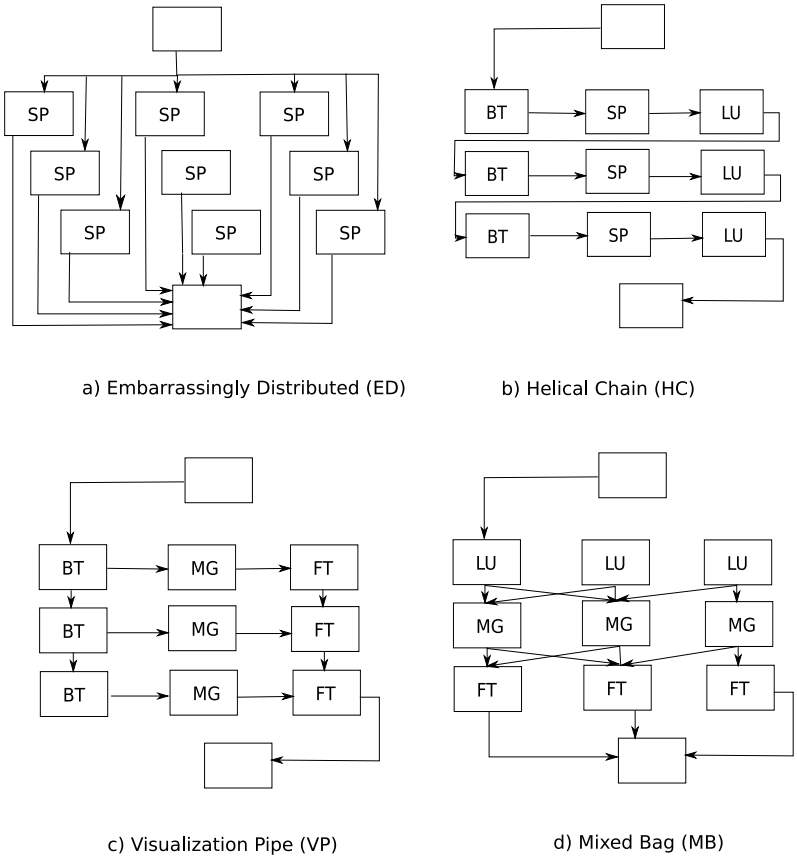


Figure 5.20: Patterns for NAS Grid

the NAS Grid Benchmark (NGB) [Fru01], and thus provide effective support for the pattern-based implementation of grid algorithms over large topologies. Then, we evaluate the approach qualitatively and quantitatively for a non-trivial extension of NGB that extends it with error recovery support in form of a checkpoint algorithm.

5.9.1 Motivation: communication patterns on grid applications

To motivate our approach we have analyzed the NAS Grid benchmark (NGB) framework [Fru01] for grid infrastructures. The NGB framework is a benchmark suite for computational grids that addresses one of the most important features of grid computing, the ability to execute distributed, communicating processes. NGB is frequently used for testing programming tools and compiler optimizations. Furthermore, it provides a real-world example of the use of computational and communication patterns in real-world grid applications.

In general terms, NAS Grid provides facilities for the benchmarking of grid applications that are based on the following four patterns [dWF04] (see Fig. 5.20): Embarrassingly Distributed (ED), Helical Chain (HC), Visualization Pipe (VP) and Mixed bag (MB).

Benchmarks are produced using the NAS Grid framework by defining graphs of nodes that represent calculations and edges that indicate how results of computations have to be calculated, ordered and passed between nodes. An instance of a benchmark is specified by a

static data flow graph (DFG). The DFG consists of nodes connected by directed arcs. NAS Grid includes an imperative and low-level language to describe such graphs by enumerating all nodes and edges. Communication between nodes is asynchronous. A DFG node receives input data from other nodes through its input arc(s); this data is used by the target node(s) to set initial conditions and to perform the target nodes' calculations. A DFG node starts its computation only after it receives all data from its predecessor(s) in the graph. After performing its calculation, it sends the computed result along all of its output arcs.

Example: Global Checkpoint Error Recovery

In order to illustrate the problems in the implementation of distributed algorithms over grid architectures and evaluate our solution to them we investigate a fundamental service in grids, global checkpoint error recovery.

Checkpoint recovery is a service that facilitates the recovery and the continuation of an interrupted computation. This service is essential for large, long-running computations to minimize downtime and other costs incurred by system or applications failures that stop the computations. A checkpointing service periodically saves the state of the applications and the manipulated data. For a distributed application, a distributed checkpoint is a set of local checkpoints, one from each process constituting the overall distributed computation. In this situation, the service must ensure the global consistency of the captured state. Consequently, a checkpointing service has to inspect and modify the local computations in an invasive manner.

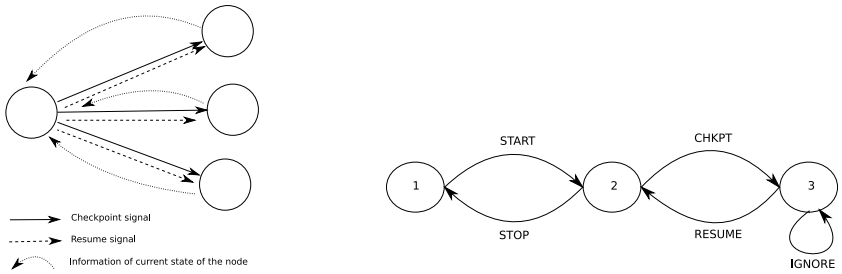
In grid environments, global checkpoint recovery is particularly important to facilitate migration and continuation of incomplete computations in the context of temporarily unavailable resources. However, for large scale applications, checkpointing is subject to two specific problems. First, some specific applications embody theoretically and experimentally validated algorithms, whose correctness must not be endangered through source code modification. In this situation, a generic approach that does not require any code modification can be used but impacts the memory footprint and thus frequently is not viable for performance reasons [SS98]. Our solution that uses an aspect based approach allows a checkpointing service to be implemented while transparently modifying the interaction between legacy components with a negligible impact on the memory footprint and other performance characteristics. Second, defining grid algorithms concisely over large-scale topologies depends on the application at hand and is, for instance, very error-prone and tedious using NGB's low-level means for topology definitions. In the case of global checkpoint recovery, a complete representation of the communication state is needed to ensure the necessary global coordination for the capture of a coherent state: a coordination algorithm that may be complex and specific to the application communication model thus has to be developed. Invasive patterns support the concise description of modifications to computations and communications between components and thus enable, in particular, checkpointing to be added modularly to the NGB.

5.9.2 Evaluation

To evaluate our approach we have implemented an extension to the NAS Grid benchmarking framework by adding checkpointing support. This implementation coexists with the native communication mechanism of the NAS Grid framework (we used its Java RMI instance); AWED is exclusively used to implement the checkpointing concern. This concern uses a dif-

ferent distributed topology than that directly implementable by the different configurations of NAS Grid. Figure 5.21a shows the topology structure and distributed messages of the checkpointing algorithm that we have used in the experiment. In the experiment, any node, even an external node, can generate a *Checkpoint* signal: upon reception of that signal, a node stops its current computation, stores a consistent state, sends that state to the centralized checkpointing monitor and waits for a *resume* signal. Thus, the application will use a composition of the farm and gather topologies as presented in section 5.4 figures 5.8b and 5.8c. This simple algorithm does not require synchronization between nodes but needs to weave the underlying application with joinpoints that have to be propagated in the (distributed) grid: this algorithm allows to evaluate the actual overhead of the runtime infrastructure imposed by the AWED implementation of invasive patterns. The algorithm is fully distributed and any node can serve as coordinator of the checkpointing protocol.

Figure 5.21b shows a representation of the state machine controlling the checkpointing algorithm in the distributed nodes. In the native infrastructure, we have identified two joinpoints per node that are relevant for our implementation of checkpointing. The first joinpoint (START transition) corresponds to the execution point when data from previous calculations is received by a node; the second (STOP) corresponds to a node having just sent calculation results to the next node in the calculation graph. When a checkpoint signal (CHKPT transition) is received a consistent local state (state before calculation) is stored locally and also, by our checkpointing implementation, remotely in the checkpoint data structure. Thus, after a failure, recovery will be carried out locally by each node depending on the state of the node. If it determines that it has been in the third state, it will relaunch the calculation, otherwise no specific action is needed (because the result of the last computation has already been sent).



(a) Topology (b) State machine

Figure 5.21: Topology and state machine representation of the protocol implementation for check pointing.

Figure 5.22 shows the implementation of this checkpointing algorithm using AWED. The aspect defines two local fields to store the checkpoint image. The image is created when the first event defining the `START` transition is received by the advice triggered by the pointcut `step(chkptSequence(), START)`. As a second step, the aspect waits for a checkpoint event (transition labeled `CHKPT`) or a finalization event (transition `STOP`). In case of a checkpoint event the aspect waits for a resume event (see transition `RESUME`) to reinitialize the calculation. Finally, the transition `IGNORE` ensures that terminated calculations are ignored and avoid gathering data or restarting computations in this case. This last transition guarantees that no further events are sent after a checkpoint image is captured: checkpointing thus conforms to the notion of consistency introduced in section 5.9.1 (a checkpoint is taken between reception

of data and the end of the corresponding calculation).

```
1  aspect ChkPtAsp perobject {
2   BMRequest req;
3   BMResults res;
4
5   public BMRequest requestChkPtInfo(){}
6
7   pointcut chkptSequence():
8   seq(
9   START: call(* BenchUnion.startBenchmark(..))
10        && host(localhost) > STOP || CHKPT;
11  STOP: call(* BenchServer.PutArcData(..))
12       && host(localhost);
13  CHKPT: call(* chkimpl.MainConsole.stopCalculNow(..))
14      && !host(localhost) > RESUME || IGNORE;
15  IGNORE: call(* BenchServer.PutArcData(..))
16       && host(localhost) > RESUME || IGNORE;
17  RESUME:
18        call(* chkimpl.MainConsole.startCalculNow(..))
19        && !host(localhost) > STOP || CHKPT);
20
21  before(): step(chkptSequence(), START)
22  { System.out.println("Asp:Iniciando...");
23     BenchUnion comp =
24         (BenchUnion) thisJoinPoint.getCalledObject();
25     req =
26      (new BenchUnionInspector(comp)).getRequest();
27  }
28
29  around(): step(chkptSequence(), IGNORE){
30     return new Object();}
31
32  after(): step(chkptSequence(), RESUME)
33  { BenchUnion comp = new BenchUnion(req);
34     comp.startBenchmark();
35  }}
```

Figure 5.22: Checkpoint concern implemented using AWED

Qualitative evaluation

We have first performed a qualitative evaluation by comparing how concise grid applications can be expressed in terms of the native topology configuration language provided by NAS Grid and our pattern language.

A comparison between the native NAS Grid language for the definition of DFGs (fig. 5.23) and our pattern language (fig. 5.24) shows that the abstraction of host groups we introduce make the declaration of grid topologies much more concise. A large-scale grid application is frequently composed of over 1,000 processes. Without patterns and pattern composition, the task of defining a grid application relying only on the NAS Grid language is very tedious. Typically one needs to write one line to define a node and one for the link between two nodes. Our language is concise (a few lines define groups and connect them with patterns), and it

supports pre-established properties (*e.g.*, synchronization, topology). For the sake of readability, we directly use pattern names such as `farm` and `gather`. These terms can be formally defined as syntactic sugar in forms of macros that are expanded into plain aspect definitions: `farm(G1,Afarm,G2)`, for example, becomes `G1 {aspect Afarm ...} G2` in terms of the pattern language introduced in Sec. 5.5. (In our prototype implementation, we have used a straightforward script to expand intensional group definitions, such as `G2={h1..h7}`).

```
graph: {
title: "ED.A"
node:{title: "0" label: "SPTask.A.h0"}
node:{title: "1" label: "SPTask.A.h1"}
node:{title: "2" label: "SPTask.A.h2"}
node:{title: "3" label: "SPTask.A.h3"}
node:{title: "4" label: "SPTask.A.h4"}
node:{title: "5" label: "SPTask.A.h5"}
node:{title: "6" label: "SPTask.A.h6"}
node:{title: "7" label: "SPTask.A.h7"}
node:{title: "8" label: "SPTask.A.h8"}
edge:{sourcename: "0" targetname:"1"}
edge:{sourcename: "0" targetname:"2"}
edge:{sourcename: "0" targetname:"3"}
edge:{sourcename: "0" targetname:"4"}
edge:{sourcename: "0" targetname:"5"}
edge:{sourcename: "0" targetname:"6"}
edge:{sourcename: "0" targetname:"7"}
edge:{sourcename: "1" targetname:"8"}
edge:{sourcename: "2" targetname:"8"}
edge:{sourcename: "3" targetname:"8"}
edge:{sourcename: "4" targetname:"8"}
edge:{sourcename: "5" targetname:"8"}
edge:{sourcename: "6" targetname:"8"}
edge:{sourcename: "7" targetname:"8" }}
```

Figure 5.23: Farm-Gather topology with NAS language

```
G1={h0}
G2={h1..h7}
G3={h8 }
gather(farm(G1,Afarm,G2),Acalc,G3)
```

Figure 5.24: Farm-Gather topology with pattern language

We have implemented the functionality for checkpointing and recovery using AWED by two classes and one aspect accounting for a total of 93 lines of code (LOC). Achieving the same functionality in native NAS Grid using Java RMI will require the modification of the current framework for distribution that amounts to 3939 LOC. We could also create an additional framework for the distribution, concurrency and coordination of the checkpoint functionality, but in both cases we still have to modify the original framework. It is clear that our a compact implementation facilitates maintenability and redability.

Performance Evaluation

In order to evaluate the performance impact due to our AWED implementation, we have run the NAS Grid benchmark on Grid'5000, a grid of 5,000 processing units distributed over 9 French sites. Note that the number of resources that can be allocated for an individual experiment depends of the number of request from affiliated laboratories.

In order to present the overhead of our AWED-based implementation, we compare the runtime of two different NASGrid Benchmark configurations that represent typical data-flow application topologies: HC, a fully sequential distributed topology; and FG a typical master/slave distributed topology where an initial node propagates tasks to a farm, and then results are gathered on single node. In both cases, each node is running the same component. For each experiment, we have deployed the components on two different clusters located on two different sites and run the experiment 3 times. Figure 5.25 shows the average overhead due to the AWED framework and the checkpoint service, using two different application topology and a variable amount of computing nodes. As described earlier, the checkpointing service records a consistent state of each host at two joinpoints: first, when a local node receives data from previous calculations, second when the node just sent calculation results to the next node(s) in the calculation graph.

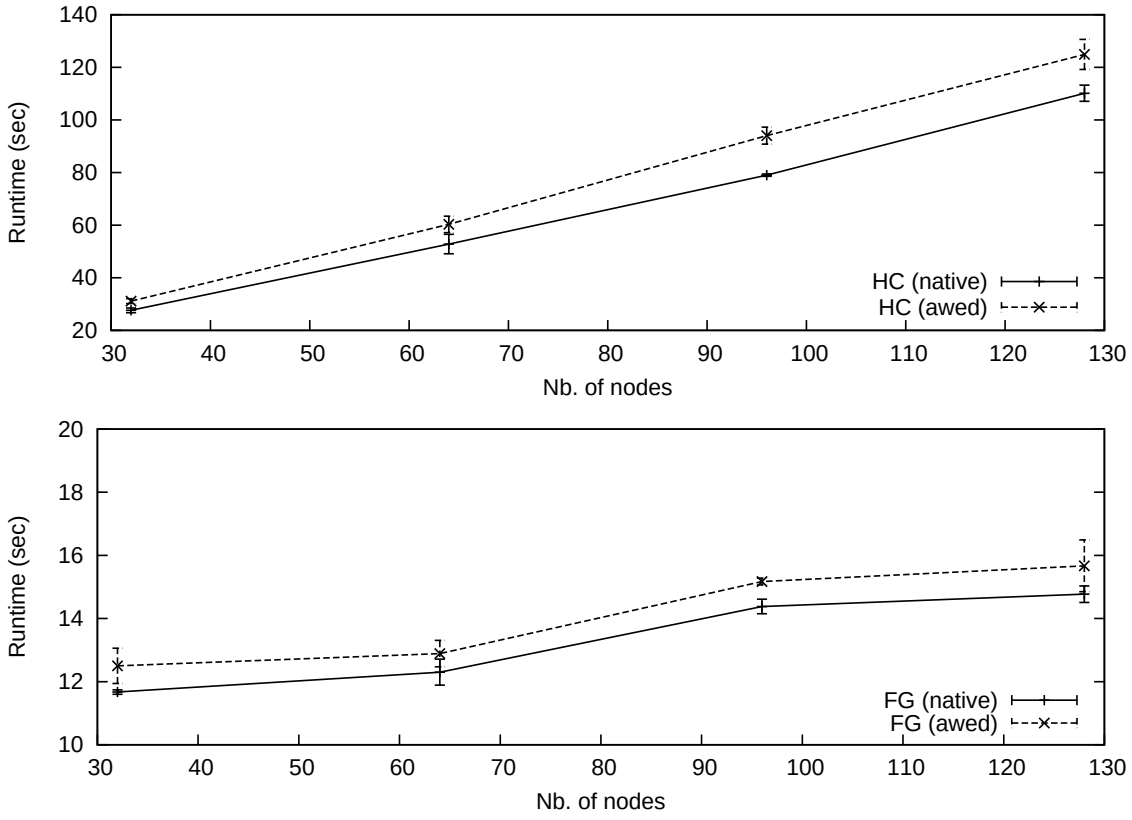


Figure 5.25: Impact of AWED implementation in NASGrid

The AWED implementation shows an acceptable (less that one second) overhead in the case of the massively parallel (FG) benchmark: in this case the runtimes of the native benchmark and the benchmark with AWED are comparable. On the fully sequential benchmark

HC, the global overhead is more important but still acceptable. This is due to the *local overhead* between each node that accumulates over sequential executions. Note that AWED's current implementation generates a distributed message for every call to `startBenchMark` in the class `BenchUnion`: thus the total number of messages including checkpointing is circa double that of the NAS Grid algorithm without checkpointing. It can be expected that further optimization in aspect weaving and message delivery lead to a smaller communication overhead.

Other approaches for grid programming

Frameworks for grid applications. Distributed applications are often built using rich middleware structures, which provide basic services for the implementation typical computation and communication patterns. In the domain of grid computing, for instance, Globus, one of the most popular middleware for grid architectures, uses the resource specification language RSL [Glo] to support the deployment of applications. In contrast to the notion of invasive patterns advocated here, computation and communication patterns have to be programmed in an ad hoc manner, in particular because RSL cannot describe connection constraints between the parts of an application that depend on execution state that is encapsulated by the distributed nodes.

Sequential Aspects for grid applications. Some research and industrial approaches have addressed the use of AOP techniques in the context of grid applications. Sequential aspects have been used to implement monitoring and management of grid applications [GPB05]. Furthermore, they have been employed to address composition in workflow systems for grid services [JVS06]. Finally, recent industrial efforts, such as the Gridgain approach [gri] claim to use AOP to enable transparent configuration and modification of grid applications. These approaches apply directly traditional sequential AOP techniques and do not explore declarative support of aspects to define and implement fully distributed invasive patterns as motivated by our research. As discussed in this section, such approaches cannot implement crosscutting concerns in distributed applications as concisely as using our proposal when such a concern has to refer to different nodes.

5.10 Discussion and future work

Software patterns have proven a versatile tool for program development. They facilitate application development and maintenance by raising the abstraction level of descriptions for software artifacts. Patterns have been very successful for sequential object-oriented applications, as well as for massively parallel algorithms. However, pattern-based approaches have been much less successful in the domain of distributed programming that are defined on irregular topologies and subject to inhomogeneous synchronization requirements. In this chapter we have identified a major reason for the difficulty in applying programming patterns to distributed applications: applications of such patterns frequently depend on information that is not locally available where the pattern is to be applied.

In this chapter we have proposed a solution: *invasive patterns*. Such patterns provide well-known computation and communication patterns (e.g., pipe, farm and gather) but also offer a built-in abstraction based on AOP for access to non-local state. We have motivated our approach in the context of JBoss Cache, a real-world infrastructure for transactional replicated

caching. We have introduced a language for defining and composing *invasive patterns* that has been implemented by a translation into AWED, a system for explicitly distributed AOP. Finally, we have evaluated our approach qualitatively and quantitatively by presenting a non-trivial pattern-based refactoring of parts of JBoss Cache.

Our proposal provides a solid basis for numerous future work. First, *invasive patterns* currently support static only topologies, but AWED supports groups of hosts that evolve dynamically. Our language could easily be extended to benefit from this mechanism. Second, our semantics is a simple translation to AWED so it offers many optimization opportunities (e.g., aspects deployment on specific hosts, pattern composition specialization). Finally, patterns raise abstraction level of software and are prime candidates for formal methods (properties to be analyzed include communication protocol compliance, absence of deadlock, topology invariants, fault tolerance).

Chapter 6

Causally ordered sequences: debugging and testing distributed middleware

Many tasks that involve the dynamic manipulation of middleware and large-scale distributed applications, such as debugging and testing, require the monitoring of intricate relationships of execution events that trigger modifications to the executing system. Such relationships, which often include events occurring on different hosts, have to be defined declaratively as well as monitored and modified efficiently. Consider, for instance, coherency of replicated data under transactional control in middleware cache infrastructures, such as JBoss Cache: in this case, the correctness of sequences of events corresponding to executions of two-phase-commit protocols involving multiple machines has to be checked. Furthermore, execution events of a distributed system frequently are of interest only if they occur as part of specific execution traces but not in the presence of different interleavings of the events that are part of those traces and occur due to non-deterministic executions. The definition of reproducible test cases, for instance, frequently requires constraints to be imposed on non-deterministic executions.

Several approaches to define such relationships among and constraints on events in distributed systems have been proposed. Such approaches include, for example, causal event relationships based on logical clocks [Lam78, FZ90, And01], data path expressions for concurrent programs [PHK91], and control-flow based event relationships [Li03]. However, such declarative means for the definition of event relationships have not been integrated into mainstream middlewares and corresponding support in current tools for the debugging and testing of distributed infrastructures is very limited. Intricate relationships between distributed events and restrictions on the interleavings of concurrent events can be directly defined in current execution environments only in terms of conditions on the execution state of individual hosts. Hence, relationships involving multiple hosts have to be expressed using complex encodings that are difficult to understand, to maintain, and result in inefficient event monitoring and execution of modifications.

In this chapter, we argue for the use of high-level abstractions for the definition of relationships between execution events of distributed systems, their modification and the control of non-deterministic interleavings of events. Concretely, we have structured the chapter as follows. First, related work is discussed in Sec. 6.1. Second, we motivate that such mechanisms

improve on current debugging and testing methods for distributed systems, in particular, real-world middleware infrastructures (Sec. 6.2). Third, we introduce corresponding aspect-based programming language support that provides declarative means to monitor and modify causal sequences of events in pointcuts and advice. We then present suitable language support (Sec. 6.3), based on existing notions of logical clocks, and a corresponding implementation (Sec. 6.4) in terms of an extension of the AWED language and system. Then, we evaluate our approach in Sec. 6.5 in the context Java-based middlewares, in particular, for debugging and testing of JBoss Cache, and ActiveMQ, the Apache message broker. We also show how current best practices for the debugging and testing of distributed systems can be improved using our approach in a practical and efficient manner. A closing discussion is given in Sec. 6.6.

6.1 Related work

Our approach for causality is based on vector clocks introduced by Mattern [Mat88] (that itself extended Lamport's approach on logical time introduced in the landmark paper [Lam78]). These results were later integrated into actual middleware for reliable distributed systems based on group communication,*e.g.*, see the Horus framework [vRBM96]. The benefits and limitations of using causal communications, in particular, the resulting overhead that is added to all communication, has been actively discussed [CS93, Bir94, SM94]. Our approach extends similar current approaches, *e.g.*, the support for causality in JGroups [Ban02] that strive for the use of limited notions of causality that are used to ensure selected properties of distributed systems. We have provided concrete evidence that expression of causal communication at the language level is useful in the presence of real-world debugging scenarios in current middleware.

Research on logical clocks was also related to research on deterministic global states in distributed applications, see *e.g.*, work by Chandy and Lamport on global snapshots [CL85] and the work on *the state machine approach* [Sch90] by Schneider. Our work address these ideas allowing the programmer to control the ordering of messages and the definition of finite state machines that are controlled by distributed events. Thus, the programmer can simulate replicated state machines that consume events, respecting a partial ordering of events.

Debugging of control-flow based relationships between execution events has been one of the main domains of application of causality and logical clocks, see *e.g.*, [PHK91, HK90, FZ90, DRGL+07, SVAR04]. Hseush et al. [HK90] and Ponamgi et al. [PHK91] have presented *Data Path Expressions* (DPE), a control-flow based debugging language for concurrent applications. The language addressed causal relations in a concurrent program. Their model for a debugger proposed a centralized process for DPE evaluation that consumes events provided by other processes instrumented by the debugger. Our sequence construct combined with the pointcut language provide similar flexibility as their theoretical language, additionally we provide a fully distributed solution with no central monitoring component. Fowler and Zwaenepoel [FZ90] have introduced causal breakpoints that are based on the idea of having consistent global states in distributed applications. When a traditional sequential breakpoint is reached their system returns to a stable state, *i.e.*, a causally consistent one with respect to the breakpoint. Our approach, though supports more sophisticated pointcuts.

More recently Sen et al. [SVAR04] proposed an algorithm for decentralized monitoring used to check violations of safety properties in distributed systems. Monitoring expressions in their approach are written in past time linear logic. Their proposal presents *knowledge*

vectors (inspired by vector clocks) and propose the Diana tool and actors as an implementation support. The approach also propose a weaving mechanism at the byte code level as we do. Their epistemic construct is similar to our `host` pointcut, which is used to predicate over the process (host) where the event occurred. Our approach provides richer expressivity because of our general notion of transition guards and allows group relationships to be expressed.

Other approaches have addressed the implementation and formalization of distributed models for debugging (*e.g.*, see [DRGL+07, MK07]). However, either they do not consider the causality concept and ordering of events (*e.g.*, De Rosa *et al.* [DRGL+07]) or, they restrict the concept of causality to the concept of distributed control flow (*e.g.*, Mega and Kon [MK07] as well as Li's work on monitoring of component-based systems [Li03]) who considers causal relationships between execution events of distributed (CORBA or COM) component-based applications that are defined in terms of control-flow relationships only. These approaches can only express a small subset of the relationships we consider. Finally, control flow relationships for the debugging using aspects have been considered only for the sequential case, notably by Chern and De Volder [CV07].

Finally, scalability and decoupling in distributed systems has been addressed using event driven approaches. Several language related abstractions has been proposed and studied, *e.g.*, see API like extensions defined in the JMS specification [HBS+02], type safety in Type-based publish subscribe [Eug07]), or even approaches addressing synchronization and business concern separation using aspect oriented techniques and linear temporal logic [MS05]. However, none of them present all features needed to support distributed debugging as presented in this chapter. Our approach adds abstractions for synchronization and distribution that scale well by means of a decentralized architecture, and provides dynamic and efficient weaving of aspect for distributed debugging.

Programming languages offers numerous features (*e.g.*, classes, mixins, modules, functors, etc.) to support separation of concerns. AOP offers programming language support for separation of crosscutting concerns. AspectJ is probably the most well-known and used aspect weaver [KH+01] it offers an aspect language and and an aspect weaver (*i.e.*, compiler) for Java but it does not provided concurrency or distributed related features. Awed [BNSV+06a] is an aspect weaver for Java that explicitly support distribution with notions such as distributed pointcuts, remote advice, asynchronous advice, etc. Our work is based on awed but invasive patterns provide a higher level abstraction that makes it easy to program and compose aspects.

6.2 Motivation

In this chapter, we argue for the use of explicit relationships between events to be used to monitor and manipulate middleware and distributed infrastructures. We claim, in particular, that control-flow based relationships, sequence relationships and events that are causally-connected, *e.g.*, with respect to a notion of logical time, are crucial in this context. In this section, we motivate these claims for typical debugging and testing tasks of distributed infrastructures.

6.2.1 Expressive breakpoints for distributed debugging

Current tools for distributed debugging, such as Eclipse and the Distributed Debugging Tool [Sof08, Fou08], apply debugging techniques for sequential programs to distributed applications. Such tools almost always employ a centralized debugging component that coordinates

execution of independent local debuggers that only support breakpoints in terms of local entities (*e.g.*, updates of local objects, local files, etc.). The distributed debugger can match local breakpoints in different machines and control the execution by, *e.g.*, stopping it and inspecting the local state of different machines. However, this kind of tools has not been widely adopted by developers, mainly because they do offer only small added value over the use of sequential debuggers on a per-machine basis.

We argue that there are three major reasons for this lack of added value:

- Lack of means for the expressive definitions of distributed breakpoints involving, in particular, control flow and sequence relationships between distributed execution events.
- Lack of means to handle non-determinism in distributed and concurrent applications.
- Inefficient implementations based on centralized architectures that are difficult to deploy on top of a distributed application.

In the following, we consider three basic debugging scenarios that frequently occur in middlewares to illustrate these issues involving control-flow relationships and non-deterministic relationships among events, especially ones involving causally-connected events (thus effectively extending discussions in recent work on distributed debugging [Li03, MK04]).

Debugging control flow

As a first example, consider a distributed application that uses synchronous remote method invocation (*e.g.*, Java RMI) for communication between three different hosts. A developer may be interested in setting a line breakpoint in one host, `H` say, that is triggered only in the dynamic extent of a (previous) method call occurring on another host `G`. Note that such debugging scenarios are based on (typically implicit) specifications of correct program behavior. *e.g.*, that an erroneous execution path is characterized by the sequence of calls `G;H` on the mentioned hosts where `H` occurs before the call to `G` returns. Using current tools, the developer has three options:

- She can apply a breakpoint to the method called on host `G` and once this match is triggered she can, at runtime, add the line breakpoint at `H`. However, in this case all subsequent occurrences of the second breakpoint are matched: identifying a specific call of interest can be very difficult.
- The programmer could pollute the original code with state information to track the necessary control flow dependencies (*i.e.*, store state information that then has to be suitably forwarded to the other hosts) and match the specific breakpoint in `H`.
- The programmer could add a breakpoint directly on the execution of `H`, match the corresponding breakpoint there without taking into account the originating control flow and decide manually what to do at each match.

Using (formal or informal) reasoning mechanisms, all of these options could be proven to correctly identify the erroneous path with respect to the specifications above. However, clearly none of these situations is acceptable, because they are tedious to implement and are highly error prone. All three represent, however, common practice with current debuggers for distributed middleware and applications.

Debugging non-deterministic event relations

Events that may occur concurrently and that should trigger debugging operations only if they are interleaved in specific ways further complicate matters. Debugging of replicated caching infrastructures, for example, may involve replication actions that originate from the same transaction but are triggered asynchronously (e.g. as part of a two phase-commit protocol). Errors often depend in this case on the order in which the replication actions are applied but the decision, as part of a debugging action, whether two actions occur in the relevant order is difficult to take if debugging processes (as is typically the case) may introduce arbitrary delays in the observation of events.

Since current debugging tools do not provide abstractions to concisely express such cases, programmers once again have to resort to manually encode and interpret distributed state by applying one of the three options introduced above. This approach becomes, however, rapidly unmanageable if many events and many hosts are involved.

Often such debugging tasks can be much facilitated by ensuring that occurrences of events obey strict ordering requirements, possibly imposing deterministic sequences of events in a previously non-deterministic systems. This is useful, in particular, in order to systematically explore possible erroneous traces. Once again current debuggers do not support such facilities, but have to resort to encodings of distributed state. Extending previous work [HK90, PHK91, Li03, MK04] that has highlighted casual relationships as a means to remedy this problem, our approach seamlessly integrates notions of causality with expressive control-flow based event relationships.

6.2.2 Test-driven development

Current techniques for the test-driven development for distributed applications are also limited by a lack of support for the expression of distributed event relationships. Distributed unit test cases, in particular, are almost always implemented by means of sequential abstractions that test conditions of distributed concerns on the local state of individual machines. For example, test cases related to replication in JBoss Cache [JBo08b] frequently use a seemingly intuitive testing scenario: a test case is defined in terms of two cache instances, such that after an operation on a source cache, the state of the second cache can be tested to compare the new and old versions. This idiom seems obvious and simple; however, it does not allow to take into account, for example, the communication behavior, such as sequences of intermediate synchronous or asynchronous calls, which obviously may strongly interfere with the cache behavior. Consequently, the definition of reproducible test cases is subject to the same restrictions as discussed above, for example, if reproducibility depends on specific interleavings of a set of concurrent events being tested (that are part of a potentially much larger set of possible interleavings).

As we show below, expressing test cases in terms of sophisticated event relations as described above (control-flow, sequence and causal relationships) also greatly facilitates the definition and application of distributed test cases.

6.3 Language support

In this section we propose a language to support manipulations and evolutions of distributed applications. It is based on the AWED system (See chapter 4): that explicitly supports

monitoring of sequences of distributed execution events that trigger dynamic modifications. This enables us to concisely express different debugging scenarios involving control-flow and sequence-like relationships between events. Furthermore, we introduce in this chapter an extension of AWED in order to support causally-related events and causal communications (based on event reordering mechanisms).

Concretely, we have integrated causal relationships and reordering mechanisms extending the AWED language for the manipulation of distributed applications using aspects with explicit features for distribution. AWED natively supports pointcuts that match sequences over (nested or not nested) method calls and trigger advice that may be executed on hosts different from that on which the pointcut matched. This resulting model of distributed programming can also be seen as an event-based model with expressive means to match execution events (through pointcuts) and trigger new functionality that modifies (including replacement of) matched execution events (through advice).

The original AWED model provides a notion of remote pointcuts for matching of sequences of execution events without support for causality relationships. As we will show, these pointcuts generalize advanced debugging facilities that have been proposed previously. The extensions for causal pointcuts and pointcuts with causality-dependent reordering augment the languages expressiveness, supporting, in particular, the causality-dependent debugging techniques discussed before and integrate smoothly with the non-causal pointcuts on the syntactic and semantic level.

In the remainder of this section we introduce the resulting language in three steps. First, we show how AWED allows a more general treatment than several previous approaches to debugging based on event sequences. Second, we provide background information on causality relationships. Finally, we extend the AWED language by means for the definition of causal pointcuts that allow logical time to be taken into account as well as pointcuts matching event sequences that are reordered according to causality relationships.

6.3.1 Distributed debugging with AWED

AWED can be applied to debug intricate relationships between execution events. It generalizes previous approaches to the debugging of control-flow based relationships between events. In this subsection we show how the original AWED model allows to handle debugging problems expressed in terms of control-flow-based and arbitrary sequence-based relationships between distributed events.

Distributed control flow

Sequences of calls that are nested within each other's control flow can be defined using the `cflow` pointcut constructor. As an example consider testing and debugging of JBoss Cache as presented in the motivation section. A concrete problem of the two-phase commit protocol consists in ensuring that remote calls to `prepare` methods are always triggered by a corresponding call at a local cache site. A remote call that has not been appropriately triggered can be caught by the following pointcut:

!cflow(**call**(∗ Transaction.prepare(..)) && **host**("source"))
&& **call**(∗ Cache.remotePrepare(..)) && **host**("target")

This pointcut matches all the calls to the `remotePrepare` method on hosts belonging to the host (or host group) `target` that are not in the distributed control flow of calls to the

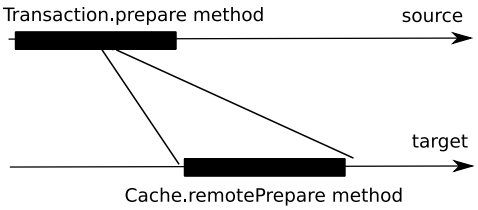


Figure 6.1: Distributed Cflow graphical representation

`prepare` method occurring at `source` hosts. Hence, a simple pointcut definition can address the complexity of a distributed control flow breakpoint. Such control-flow relationships for debugging have already been studied, *e.g.*, as part of Li's work [Li03] for distributed (CORBA and COM) component-based systems and Chern and De Volder's work on sequential control-flow based breakpoints [CV07]: we extend such approaches by supporting the notion of control flow in the presence of asynchronous and synchronous method calls.

Figure 6.1 represents graphically the concept of control flow. The parallel lines represent the processes on two different hosts `source` and `target`. The dark bars represent the execution of the corresponding method, *i.e.*, the execution of the body of the method before the method's return. Considering synchronous communication, the figure represents a call to the `remotePrepare` method that is in the control flow of a `prepare` method, thus in this case the previous pointcut definition will not match the call to method `prepare`.

Distributed sequences of events

As introduced above, AWED supports pointcuts over sequences of execution events, *e.g.*, sequences of calls that do not have to be nested into other calls of the sequence. Hence, such sequences allow the definition of more general event-based contexts than the control-flow based event sequences considered above.

In the context of the debugging of JBoss Cache, for example, a very frequent requirement consists in the definition of contexts depending on the activation state of the cache. Concretely, one may want to identify remote `put` operations (which introduce data in the cache) that occur after the local cache has been initialized and before it has been stopped. A corresponding pointcut can be specified in AWED as follows:

a1 : **seq**(start > t1,
t1: **call**(∗ Cache.start(..)) && **host**(**localhost**)> t2 || t3,
t2: **call**(∗ Cache.put(..)) && !**host**(**localhost**) > t2 ||t3,
t3: **call**(∗ Cache.stop(..)) && **host**(**localhost**) > t1)
&& **step**(a1,t2)

This pointcut defines an automaton named `a1` having three transitions `t1`, `t2` and `t3`: once started, `put` operations can occur or the cache can be stopped. Note that the `start` and `stop` operations of the cache are matched on the local host, while the `put` operations must not occur on the local host. The term `step(a1, t2)` allows an advice to be triggered relative to a specific transition `t2` of the automaton. At the first line `start > t1` defines that the initial transition is `t1`. The expression `t1: pointcutDef > t2 || t3` is interpreted as follows: if `pointcutDef` matches the current event, then the automaton is now ready to accept an execution event as defined by `t2` or `t3`. Figure 6.2 shows the graphical interpretation of the defined automaton.

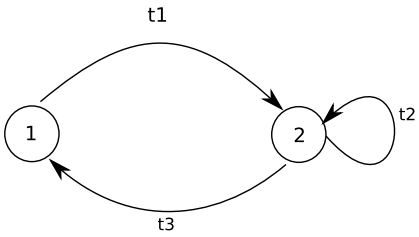


Figure 6.2: Graphical representation of a start-action(s)-stop automaton

The expressive power of our approach is mainly determined by the expressivity of our pointcut language. AWED basically provides regular pointcuts. An extension by guards on transitions of the corresponding finite-state machines, thus providing a turing-complete pointcut language, is however unproblematic (and is provided as part of the existing implementation). This feature would also allow to directly characterize concurrent and timed events. By explicitly providing regular pointcuts, existing analysis techniques of, *e.g.*, deadlocks using model checking of distributed and concurrent systems, should be applicable. This is, however, subject of future work.

A second element determining the power of our approach is the granularity of events that can be referred to by pointcuts. We have restricted the pointcut language deliberately to method calls: a more fine-grained event model that would allow, e.g., to refer to the evaluation of subexpressions of arithmetical expressions (that are supported by some aspect approaches) could incur considerable execution overhead and are less relevant for the debugging of middlewares.

6.3.2 The case for causality relationships

Sequence pointcuts in AWED do not guard against problems of the underlying communication network, in particular concerning message delivery such as inversion of sent messages due to random delays in message transmission. The previous sequence pointcut involving `start`, `put` and `stop` on JBoss Cache events is unproblematic in this respect since message inversions resulting in `put` operations outside the ordinary operating conditions of cache can be easily filtered out by additional pointcuts if necessary. In other cases, *e.g.*, inversion of bank deposits and withdrawals, such problems would however wreak havoc.

Generally, AWED's automata-based pointcuts are therefore subject to two problems:

- They may not match valid sequences of events that happen to arrive in the wrong order at the host where the sequence is to be matched.
- They may match wrong sequences that stem from events that occur at different hosts in the wrong order but whose order has been inverted, *e.g.*, because of message delays, at the host where the sequence is matched.

An AWED developer has to take care in order to avoid these problems: either by the careful definition of pointcuts and manual synchronization of distributed executions or by ensuring that additional constraints on the base application's semantics exclude them. The next subsection proposes new language constructs to enable pointcuts to directly support causality relationships and ordering constraints of messages.

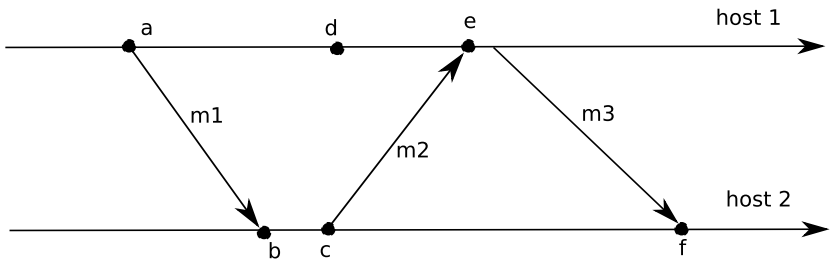


Figure 6.3: Graphical representation of the *happened before* relation.

6.3.3 Background: logical time and causality

Determining the order in which different events occur in a distributed system constitutes the essential problem underlying the limitations of general sequence pointcuts as introduced above.

Much research work has been done on orderings of execution of distributed events starting with Lamport's landmark paper [Lam78] on the use of *logical time* to determine and ensure orderings among events. Basically, such approaches exploit the property, that in order to decide how two events are ordered in a distributed system, it is not necessary to have knowledge of the physical time of their respective occurrences. Instead, one can use a logical clock that is maintained locally by each process and that is only synchronized between interacting nodes upon arrival of messages. Furthermore, (logical) clocks do not have to agree on the physical time of events but only on a (partial) order in which events were emitted.

Happened before relation

We have based our extension of the AWED model on the definition of logical time in terms of Lamport's "happens before" relation: two events a and b are related by $a \rightarrow b$ (a happens before b) if one of the following holds:

1. a and b are events in the same process and a occurs before b.
2. a is the event of sending a message from one process and b is the event of receiving the same message by another process.
3. a, b and c are events, such that $a \rightarrow c$ and $c \rightarrow b$ hold, *i.e.*, the happens-before relation is transitive.

Note that the happens-before relation can be seen to express a notion of *causality* meaning that, if $a \rightarrow b$, that occurrence of b would not have happened if a had not occurred. (This is obviously only an approximation to causality in the sense that a could be the underlying reason for the occurrence of b.) Conversely, $x \not\rightarrow y \wedge y \not\rightarrow x$ ensures that x and y occurred concurrently and may not have caused each other. Since logical time does not bear any direct relationship to physical time, concurrence in this sense does, however, not imply that x and y occurred at the same (physical) time or even during a sufficiently small time interval, but only that one event did not causally influence the other one.

Figure 6.3 illustrates the definition of the *happened before* relation. In the figure physical time flows form past (left) to future (right). The horizontal lines represent processes in particular hosts, the dots represent events, and the diagonal lines are messages. The figure

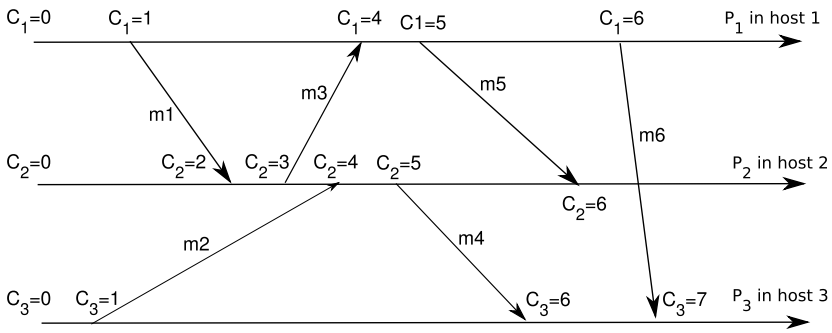


Figure 6.4: Lamport's scalar logical clocks.

shows two processes in hosts `host1` and `host2` respectively, six events (`a, b, c, d, e, f`), and three messages (`m1, m2, m3`). According to 1, in the definition of the *happened before* relation: in *host*1 a happened before d ($a \rightarrow d$) and d happens before e ($d \rightarrow e$); similarly in host *host*2, we found that b happened before c ($b \rightarrow c$) and c happened before f ($c \rightarrow f$). Because of message sending and reception (see 2 in the definition of *happened before relation*) a happened before b ($a \rightarrow b$) and c happened before e ($c \rightarrow e$). Now using transitivity (see 3 in the definition of *happened before*) b happens before f ($b \rightarrow f$); and e happens before f ($e \rightarrow f$). Finally, b and d are concurrent ($b \not\rightarrow d \wedge d \not\rightarrow b$); as well as c and d ($c \not\rightarrow d \wedge d \not\rightarrow c$).

Logical clocks

Logical time and causal relations are formalized and implemented using the notion of logical clocks. A logical clock is a function C_i in process P_i that assigns a value $C_i(e)$ to any event e occurring in process P_i. The global clock value for event e is $C(e)$ where, for all i, $C(e) = C_i(e)$ if e occurs in P_i. Additionally for any two events x and y, if $x \rightarrow y$ then $C(x) < C(y)$ (called the *Clock condition*).

A large number of different functions have been proposed for the definition of logical clocks that support (or not) different forms of causality relations. Lamport's original notion of logical clocks has been defined as scalar-valued functions. They can be implemented using a counter that has to be incremented before the occurrence of any event in a process. The clock value can then be sent together with distributed messages to implement global synchronization algorithms for distributed applications, *e.g.*, to implement mutual exclusion, in a completely distributed fashion without any global knowledge. However, the limited information transmitted by this kind of clocks does not allow causality information as introduced above to be deduced.

Figure 6.4 shows a distributed system supporting the *happened before* relation by means of an implementation of Lamport's scalar logical clocks. The figure shows a system of three distributed processes running on three different hosts (`host1`, `host2`, `host3`). Each process has an independent clock that is incremented after each event in the process (note that clocks speed is different for each process). Additionally, six messages are exchanged in the system.

Each clock C_i assigns a number to each event so that the *clock condition* is respected. Thus if events a and b occur in the same process P_i and $a \rightarrow b$ then $C_i(a) < C_i(b)$. Similarly, if a is the sending of a message in process P_i and b is the reception a that message in process P_j then $C_i(a) < C_j(b)$. To support this condition the system adds 1 to each clock after each

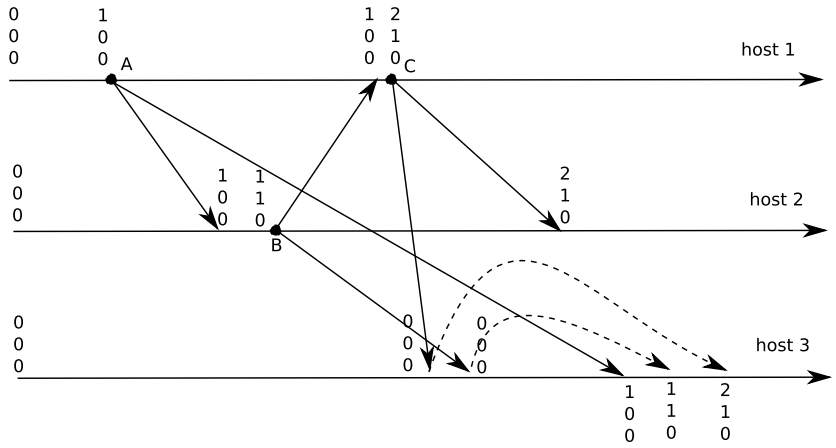


Figure 6.5: Graphical representation of the behavior of a distributed system implementing causality with vector clocks

event, time stamp each message with the new clock value, and synchronize clocks upon arrival of a message adding 1 to the maximum between the time stamp and the current clock value.

For example, when message $m1$ is sent from P_1, C_1 is increased to 1, and the message is time stamped with that value. Once the message arrives to P_2 the clock there is synchronized to $C_2 = max(0, 1) + 1 = 2$. We can follow all the message paths and check how clocks are synchronized, for instance if we follow the path $m1, m3, m5$, the clocks are synchronized successively as follows: $C_2 = max(0, 1)+1 = 2$, $C_1 = max(1, 3)+1 = 4$, $C_2 = max(5, 5)+1 = 6$. In the system the *happened before* relation is respected, thus if $a \rightarrow b$ then $C_i(a) < C_i(b)$. However, the inverse is not true, we can't sat that if $C_i(a) < C_i(b)$ then $a \rightarrow b$. Hence, as stated before, this lack of information does not allow the causality information to be inferred from the clock values.

Vector clocks

To solve the problem of causality between events in a distributed system, vector clocks have been proposed by Mattern [Mat88]. Vector clocks do not use a counter per process but associate, for each process, a vector of counters for each other processes of the distributed application. Thus, each node will dispose of information of the time at all other hosts. The following extension of the AWED model is based (as far as the matching semantics and its implementation is concerned) on vector clocks.

Previous uses of logical clocks were focused in the problem of having a deterministic and complete representation of the distributed state of a distributed application. A lot of work around this idea has been published, *e.g.*, [CL85]. They have been also used to model mutual exclusion [Lam78], high level debugging by means of behavioral patterns [BW83], and to address the problem of global predicates [TG98].

We are going to use a simple example to give an intuitive notion of causality as supported by vector clocks. Figure 6.5 present a system with three hosts that exchange messages and use vector clocks to maintain causality. Physical time flows from left to right. All vector clocks start with a vector of value (0,0,0). Each time a message is sent the local vector clock is augmented with one. For example when the event `A` is emitted the clock in `host 1`

```
// Pointcuts
Seq       ::= Id: SeqCons({Step})  | step(Id,Id)
SeqCons   ::= seq  | seqCausal  | seqCausalOrder
Step      ::= [[!]causal  | conc]Id: Pc →Target
```

Figure 6.6: AWED with causal pointcuts

is augmented to (1, 0, 0) the new value is also attached to the message being sent. Upon arrival of a distributed message each host calculates a new value for its vector clock as follows: assuming that Vc_i is the vector clock of host i and Vcm is the vector clock in the arriving message, the new vector clock is $Vc_i[k] = maxVc_i[k], Vcm[k]$. Note that the new vector clock is calculated only if the arriving message is causally the next expected message. To determine this the host should compare his own vector clock with the arriving vector clock. If the host j sends a message to the host i, then, the host i will only process the message if $Vcm[j] = Vc_i[j] + 1$ or if $Vcm[k] \leq Vc_i[k]$ for all $k \neq i$. With this conditions a host can now causally order the messages and accept them in the correct order. In the figure, host number three delays the acceptance of messages `C` and `B` to process them in the right order. A more detailed information about vector clocks can be found in [Mat88].

6.3.4 AWED with causal pointcuts

Our approach to causal pointcuts is based on vector clocks [Mat88]. These clocks can be used to enforce causal relations between events and implement causal communication by reordering events. We now show how we have integrated these notions into AWED.

Causal sequences without reordering

To extend AWED with causal information, without including reordering of messages, we have introduced a new sequence constructor `seqCausal` and two transition modifiers, `causal` and `conc`, see Fig. 6.6. The two modifiers respectively ensure that the labelled transition is causally related to or concurrently executed with respect to the transitions leading to the start state of the labelled transition. The constructor `seqCausal` is syntactic sugar for sequence pointcuts whose transitions are by default labelled as `causal` unless they have been explicitly declared using `conc` to execute concurrently.

As an example let us consider the following pointcut definition:

```
a1 : seqCausal(causal s1: call(* Cache.prepare(..)) && host("source") > s2,
               conc s2: call(* Cache.commit(..)) && host("target") > s1)
     && step(a1, s2)
```

This sequence matches a `prepare` event in a JBoss Cache transaction, followed by a `commit` only if it is *not* causally related to the `prepare` event. Then the following `prepare` event is matched only if it is causally related to the previous matched `commit` event. This pointcut can therefore be used to test for unexpected calls to commit methods. As we show in the evaluation section, Sec. 6.5, this pointcut is useful, among others, to test for real bugs that have affected the JBoss Cache infrastructure.

Causal pointcuts with reordering

Causal pointcuts without reordering only enforce that only causally-related events are matched but they do not ensure all sequences will be matched.

To resolve this second problem, we harness the property — already demonstrated by Lamport's totally ordered broadcast operation [Lam78] — that logical time values cannot only be used to test for causality relationships but that they also support the reordering of messages that arrive at a host in the wrong order. To allow reordering according to causal relationships, we have extended AWED with a third sequence pointcut constructor, `seqCausalOrder` that ensures that all causal relations are matched by, if necessary reordering, incoming events. Its semantics ensures that each event is delayed to wait for the event that precedes it causally.

As a concrete example, the following pointcut can be used to ensure that `commit` operations are correctly interleaved with `prepare` operations:

```
a1 : seqCausalOrder(
        t1: call(* Cache.prepare(..)) && host("source") > t2 || t3,
        t2: call(* Cache.commit(..)) && host("target") > t1,
        t3: call(* Cache.prepare(..)) && host("source"))
     && step(a1, t3)
```

Indeed, a web cache repeats sequences of `prepare commit`. So, two `prepare` should never occur in a row (transition `t3`): an error should be reported in this case. In order to prevent reporting of spurious errors (*e.g.*, when a `commit` occurs before `prepare` but is monitored after it) the messages must be ordered as specified by `seqCausalOrder`.

Note that this construct requires a larger overhead than the one without reordering. In particular with the previous construct the events are consumed as soon as they arrive, and causality is only an additional test defined by the `causal` and `conc` labels. In the case of causally ordered sequences, messages are delayed and processed only once all the causally preceding messages are received. The `causal` and `conc` labels are automatically supported in the totally ordered construct (they do not pose an additional overhead).

6.4 Implementation

In this section, we present how distributed aspects with support for causal events and message reordering have been implemented by extending the non-causal implementation of the AWED system (see chapter 4). Note that while we present a Java-based implementation (and an evaluation of Java-based middlewares in the following section), conceptually our approach is not tied to Java. The Arachne aspect system, for instance, features (non causal) regular sequence pointcuts for C applications and has been applied to the modification of network protocols used for the communication in distributed systems [DFL+05].

In the following, we first present the overall architecture of the resulting system. Second, we discuss how AWED can be used to test causality on distributed infrastructures that have not been prepared for the provision or use of causality information. Third, we discuss the implementation of the framework that supports causal finite state machines to support causal sequences without message reordering. Finally, we will present the mechanisms for message reordering that were included to support the pointcut construct `seqCausalOrder`.

6.4.1 AWED architecture

AWED is a dynamic aspect language that weaves aspects with classes at load time and allows aspect deployment and undeployment at execution time. Its implementation presents an optimized partially evaluated interpreter for distributed aspects. Figure 6.7 shows the overall architecture, *i.e.*, its compilation chain and the main structures of its runtime framework. In the top left part of the figure we can see how the application and aspect code is compiled into Java bytecode. The bytecode is then read by AWED's instrumentation and transformation framework at load time, producing a version of the application that is instrumented at the necessary joinpoints (here a subset of the method calls). When executing the instrumented application, and once it reaches an instrumented joinpoint, the application dispatches joinpoint notifications to the `Registry` framework that takes care of the recognition of distributed sequence pointcuts. This framework passes the joinpoint notification to each aspect instance, that, in turn, evaluates each joinpoint to match pointcuts and to apply advice. An AWED runtime framework, including a registry, is running at runtime on each logical host, *i.e.*, JVM. In order to support remote pointcuts each registry, *i.e.*, each JVM, communicates joinpoint notifications to the other JVMs using an extension of the JGroups framework [JGr08], one of the most popular Java-based middleware for group communication. This part of the infrastructure contains all necessary support for non-causal event relationships, in particular remote regular sequence pointcuts.

In figure 6.7, we have also detailed the two main extensions incorporated to the runtime framework in order to support the causal constructs. First, the communication framework (see the box labelled "JGroups extension" in the figure) has been extended to support causality-supporting protocols. The extended JGroups component uses the original JGroups framework augmented with specific protocols for causality. In the figure we show a traditional protocol stack that supports different protocols, including the User Datagram Protocol (UDP). The protocol stack shows, at the top, the `Causal AWED` protocol. This protocol can be any of two new protocols that we have implemented. Second, the pointcut class hierarchy (see the class diagram for causal pointcuts highlighted in magnifying glass in the figure) has been augmented by support for causal sequence-based aspects, concretely by support for causal pointcuts with or without reordering and a notion of transition guards. In the following we present both extensions in some more detail.

Causality-supporting protocols The two new protocols that support causality do not modify actual communication, but just handle causality and delegate actual communication to the other protocols in the protocol stack. The first protocol that we have implemented is the `Causal tags + clock increase` protocol. This protocol tags the distributed messages with a vector clock time, and will calculate the value of the new vector clock times at a host upon arrival of new messages. This protocol can be used to detect causal relations, but it can not be used to impose causal ordering of messages. The second protocol that we have implemented is `Causal tags`. This is a more lightweight protocol that tags messages with vector clock times but does not update the vector clock. This protocol can be used with specialized adapters to add causality information to distributed infrastructures and applications that have not been aware of causality information in the first place.

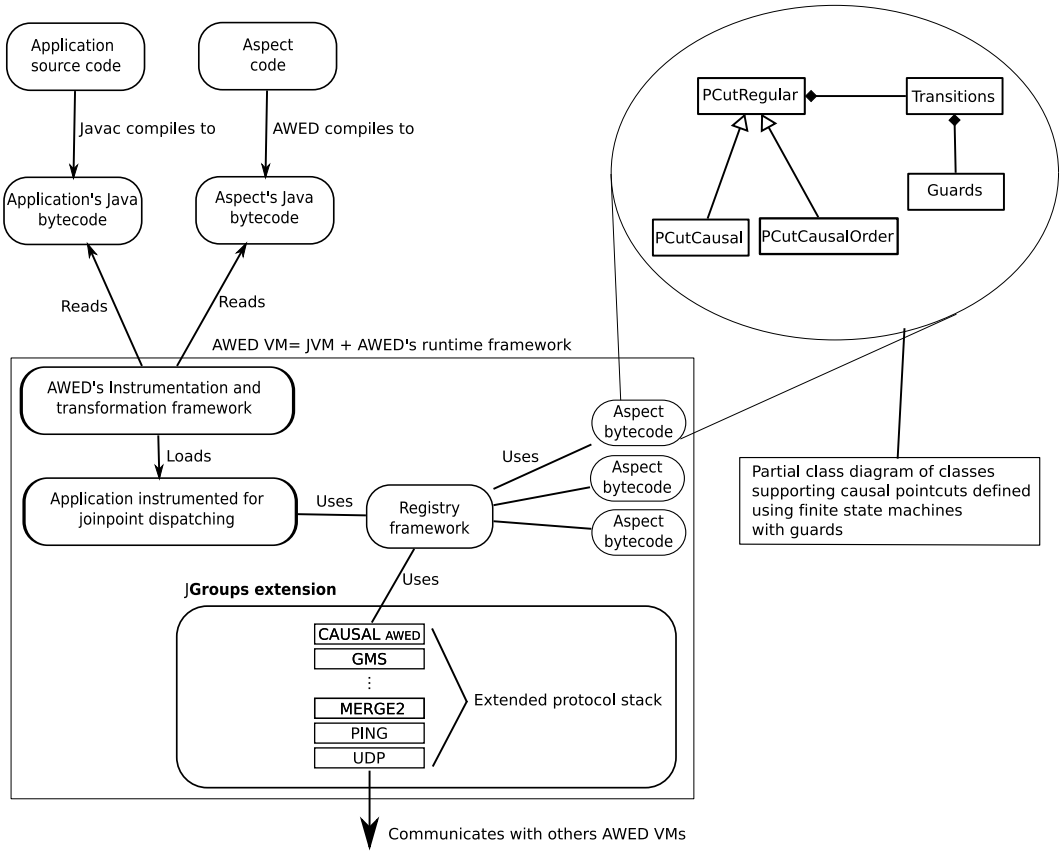


Figure 6.7: AWED architecture.

6.4.2 Adding causality to non-causal distributed applications

Most distributed infrastructures and applications do not implement causality natively. Adapting such applications to support causality typically is very cumbersome and error prone. To avoid this problem, we propose specialized adapters that can be used to instrument causality transparently in legacy applications. To prove that this is a feasible solution we have implemented an adapter for RMI based applications, thus covering a wide spectrum of distributed Java applications. This adapter is realized using Java's notion of customized sockets.

The adapter basically implements a mechanism similar to that provided by the `Causal tags + clock increase` protocol. Thus, each message in the legacy application is now tagged with a vector clock and a local vector clock is updated upon arrival of each RMI message. This connector can be combined with the AWED framework that is running the `Causal tags` protocol to detect causal relations in the legacy application. This deployment method does not need any particular modification of the legacy application. To use the specific connector, the programmer just specifies an option for the JVM when invoking AWED.

Causal sequence constructs with guarded finite state machines

In order to implement the causal sequence construct as presented in section 6.3 we have modified the compiler and the runtime infrastructure of the previous non-causal execution system of AWED. The previous AWED system has already used finite state machines to support regular sequence pointcuts. The corresponding implementation evaluates each join point and, depending on the current state of the automaton, accepts or rejects a joinpoint. In case of acceptance, a state transition is executed before executing the advice. We have extended this model to support guards. Thus, at compile time the state machine is constructed with specific guards, mainly to support the causal tests required by causal relationships expressed using the `conc` or `causal` transition modifiers.

At runtime, the new execution system includes two major extensions. First, before accepting or rejecting a joinpoint, the state machine evaluates the corresponding guard, *e.g.*, the causal information of the current joinpoint, and if the guard is satisfied the joinpoint is evaluated. The second modification address the management of vector clocks: evaluation of causal regular sequences has to compute a new value for the vector clock each time that it accepts a joinpoint. This approach has a major benefit compared with other frameworks implementing causality: finer grained control over events tagged with vector clocks and, as a consequence, less performance overhead.

Causally ordered sequences

To implement the causal sequence construct with reordering we have further extended the automata-based pointcut recognition component. Each such component now has its own vector clock that is advanced each time a message is processed (including messages not in the alphabet of the state machine). To address reordering, the state machine uses a delay queue where it stores the messages that do not arrive in the right (causal) order. The messages in this queue are causally ordered but not necessarily consecutive. Upon arrival of a new message it gets evaluated: if it is accepted and if the message causally is the next message with respect to the vector clock of the state machine, it is processed and the first message in the delay queue is evaluated again.

```
1  private void performTest() throws Exception {
2    // repeat the test several times since it's not always reproducible
3    for (int i = 0; i < NUM_RUNS; i++) {
4      if (exception != null) { // terminate the test on the first failed worker
5        fail("Due to an exception: " + exception); }
6      // start several worker threads to work with the same FQN
7      Worker[] t = new Worker[NUM_WORKERS];
8      for (int j = 0; j < t.length; j++) {
9        t[j] = new Worker("worker " + i + ":" + j); t[j].start(); }
10     // wait for all workers to complete before repeating the test
11     for (Worker aT : t) aT.join(); } }
```

Figure 6.8: Deadlock detection test case method

Finally, a note on the scalability of our approach: Concerning scalability of the pointcut matching, the principal property is that the AWED architecture (cf. Fig. 4) does not impose any centralized control, in particular, for the monitoring of pointcuts that involve causal relationships. The other components of the AWED architecture (principally matching of other pointcut types and execution of remote advice) do not require central control either as discussed as part of chapter 4.

6.5 Evaluation

In this section we present a qualitative and quantitative evaluation of our approach using JBoss Cache [JBo08b], and ActiveMQ [sf08a], a message broker implemented by the Apache foundation [sf08b]. First, we analyze a non-trivial test case for JBoss's replicated caching and show that aspects based on control-flow and causal patterns significantly improve the corresponding debugging and unit testing tasks. Second, we evaluate the performance of our prototype implementation in a two-fold manner. A series of micro-benchmarks provides evidence that our implementation supports regular causal sequences with no to reasonable small performance overhead. Finally, in order to provide concrete evidence that we meet the objectives set out in the motivation, we compare the use of AWED's use of sophisticated regular causal sequences to the use of the Eclipse debugger as a popular tool for the debugging of distributed Java applications by means of loose coordination of per-host debugging sessions.

6.5.1 Qualitative evaluation

In the following we present a qualitative evaluation of our approach involving debugging and testing scenarios for two Java-based middlewares, JBoss Cache [JBo08b] and Apache's ActiveMQ [sf08a].

Deadlock testing in JBoss Cache.

In JBoss Cache (Ver. 2.0.0GA) the method `performTest` of class `ReplicatedTransaction-DeadlockTest` (see Fig. 6.8) implements a test case to detect a deadlock bug. The test case uses two caches, actions on the first cache are replicated onto the second cache by means of the replication framework. The method triggers multiple workers in multiple threads. Each worker starts a transaction, puts a value in the cache (all workers use the same memory

```
1  pointcut deadlock():
2    s1:seqCausalOrder(
3      tPrep:
4        call(* ReplicationInterceptor.runPreparePhase(..)) && host(src) > tCommit || t2ndPrep,
5      tCommit: call(* PessimisticLockInterceptor.commit(..)) && host(targ) > tPrep,
6      tSecondPrepare: call(* ReplicationInterceptor.runPreparePhase(..)) && host(src)) &&
7    step(s1, tSecondPrepare);
```

Figure 6.9: Pointcut for deadlock detection in a synchronous transactional cache.

```
1   pointcut prepare(): call(* ReplicationInterceptor.runPreparePhase(..)) && host(src);
2   pointcut commit(): ... && call(* BaseRpcInterceptor.replicateCall(..)) && ...
3
4   pointcut generateDeadlock():
5     s1:seqOrderedCausal(
6       tPrep : prepare() > tCommit || t2ndPrep,
7       tCommit : commit() > tCommit || t2ndPrep,
8       t2ndPrep: prepare() );
9
10  before(): generateDeadlock() && step(s1, tCommit) { while(block){ Thread.yield(); } }
11  after(): generateDeadlock() && step(s1, t2ndPrep) { block=false; }
```

Figure 6.10: Aspect ensuring the generation of the buggy behavior for deadlock detection.

position in the cache) and commits the transaction. The test has to be repeated a number of times (first `for` block in the figure) since it can't be reproduced easily. The original bug occurred when a worker, after a successful prepare phase of the two phase commit protocol, commits a transaction and releases the lock over the source cache after the local commit but before completing the final commit phase with the remote caches. In this case, other workers may interleave their transaction operations, in particular, acquire the lock at the same cache position and thus preclude the first transaction to terminate its remote commit phase, thus entering a deadlock situation, because no worker can acquire all necessary local and remote locks anymore.

A programmer dealing with that bug faces tree problems: (i) how two reproduce the problem, (ii) how to debug it and (iii) how to write a suitable test case to identify it in the future. To deal with the first problem, the code shown in Fig. 6.8 triggers several threads that execute transactions concurrently, hoping for the bug to be reproduced. This approach is subject to several problems, in particular, that a unit test session could pass over the bug without noticing it. Regarding the second problem, as part of a corresponding debugging session a programmer would have to apply a breakpoint either to the line for remote `prepare` or in the line that throws the corresponding exception. In the first case the debugger will stop on each prepare (buggy or not). In the second case it will, eventually, stop only on an error of one of the threads. Then, depending of how threads are scheduled, it could stop the application(s) in a buggy state or in a correct state, because the other action could have or had not enough time to complete the transaction. Additionally, the programmer could perform many runs without reproducing the bug. A test case for this bug is, of course, subject to all the problems detailed above.

Using our approach we can improve on the three development scenarios: debugging, unit testing and bug reproduction. Fig. 6.9 shows a pointcut that can be used to define a breakpoint that will occur only if the bug appears. The pointcut implements a sequence (*i.e.*, finite state

machine) with three states and three transitions. The first state accepts a call to the method `runPreparePhase`, from the `ReplicationInterceptor` class in the cache that belongs to the `source` group (`source` and `target` are dynamic groups that can be handled using AWED). Once such a method is received, the state machine changes its state to a state that accepts `tCommit` transitions and `tSecondPrepare`, the latter representing a prepare operation issued by another worker. If the target cache receives a `tCommit` message, the normal behavior, it returns to the first state. Finally, if the sequence detects after the first `tPrepare` message a `tSecondPrepare` message on the `source` cache, the state machine recognizes a deadlock state. Note that the sequence definition must be ordered causally in order to ensure that the events will be detected in the correct order in any distributed execution.

AWED's regular causal pointcut definitions can also be helpful for bug reproduction and unit testing. The main problem with current test case definitions, such as that introduced above, is that it is of haphazard nature, *i.e.*, it does not always allow to reproduce the bug situation. Figure 6.10 shows an excerpt of code from an aspect that will interact with the original test case of Fig. 6.8 to impose the desired order of events in the presence of only two workers. The aspect excerpt includes the definition of a state machine that matches a call to the method `runPreparePhase`, which means that the corresponding transaction has acquired the lock and is going to broadcast a prepare message to the target cache. Then, if it detects a call to the `replicateCall` method having as parameter a commit method call, a before advice will suspend the current thread until another `runPreparePhase` is detected. A buggy implementation will allow this reordering of events, a correct implementation will produce a lock-timeout exception because the cache node will be locked by the second transaction.

Debugging ActiveMQ

We have also performed experiments over the Apache project's ActiveMQ message broker [sf08a] that is used, *e.g.*, for the integration of enterprise information systems. From an analysis of the list of the 359 open issues in ActiveMQ's bug tracking system as of Aug. 2008, we have found six issues classified as *blockers*: at least four of these are caused by the wrong ordering of events or messages. Similarly, out of the 13 messages classified as *critical* at least five are related to message or event ordering. We have successfully woven causal aspects on ActiveMQ. To test the applicability of our approach we have debugged a use case regarding a deadlock situation in a configuration setting with four brokers and a use case involving the wrong ordering of repeatedly delivered messages in the context of transactions session with roll back. In both cases we have successfully defined simple pointcut definitions that exactly test for the corresponding error situations. These tests provide evidence that our approach, in particular the AWED system, is applicable generally to Java-based middleware. Finally, as for JBoss Cache, these debugging experiments have incurred only minimal overhead in both the Java client and the ActiveMQ broker.

6.5.2 Micro-Benchmarks

We have run performance tests of our implementation using the performance framework of JBoss Cache. This framework allows to run multiple performance test over cache configurations. It provides several features including centralized reporting and pluggable tests. In all the cases we have used the Web-Session simulation test. This test case simulates the usage and replication of http sessions objects in a cluster of application servers. The tests were per-

formed in a cluster of 4 nodes. Each node was equipped with a double core AMD Opteron 250 (2400 MHz) processor in 32 bit mode, 4 GB of memory and a 1 GB network interface. We have performed several tests over the specific protocols developed for AWED and its runtime. In the following, we first present the test case scenario, then the tests performed the protocol implementations to show the difference of overhead compared to the native JGroups protocols. Finally, we test the full runtime performance of our approach comparing it with a session of remote debugging in Eclipse 3.2.

The *test case scenario* we have used is the default Web-Session simulator of the JBoss Cache framework that basically simulates the interaction of a replicated http session in a cluster of application servers. This test can be parameterized on the number of requests and the ratio of reads to writes requests. For each test, we specify below what protocol configuration stack we have used and what parameter values have been used. Performance is measured by the number of requests processed per second on each node. In the performance tables given below we provide the average number and standard deviation of the requests processed per second. The average and the standard deviation values have been obtained from eight different individual runs.

AWED's causal protocols vs. JGroups native protocols. We have developed two protocols that extent the set of protocols available in JGroups: the first one includes event tagging with vector clocks, increase of local clock and no reordering; the second protocol implements event tagging with vector clocks and no reordering. We have evaluated the performance of the extended protocols developed to support causality in AWED. To test only the protocol implementation and not to pollute the tests with framework overhead, we have used JBoss Cache (JBC) and the performance framework only. To this end, we have compared four different protocol configurations: (i) the performance of JBoss Cache with a standard, non-causal, configuration of its communication protocol stack (denoted `Normal` below), (ii) the causality protocol `Causal` natively provided by JGroups and (iii) our new protocols `Causal tags` and `Causal tags + clock increment`.

Protocol	Requests per second			
	20% writes		80% writes	
	Average	Standard dev.	Average	Standard dev.
Normal	63,350.23	7,004.93	58,033.77	9,792.51
Causal	60,961.14	11,867.69	53,814.05	7,085.89
Causal tags + clock inc.	52,107.34	27,790.92	53,463.53	7,310.65
Causal tags	60,396.03	7,420.05	59,487.43	7,405.64

Table 6.1: Test results of 100.000 requests with respectively 20% and 80% writes

Table 6.1 shows the results of several test sessions in our cluster. The first set of sessions was performed with a ratio of 80% reads and 20% writes over 100.000 operations (left part of the table) and the second set of tests considers a ratio of 20% reads and 80% writes (right part of the table). Each node in the test executes 100.000 requests and only the writes are replicated to the other members. The data shows that in both cases the `Normal` protocol and the `Causal tags` protocol presents the best performance average, as expected, since in the former no causal relations at all are involved and the latter is just a tagging protocol with no other additional actions. For the test with 20% writes, the `Causal` protocol (full causality, *i.e.*, vector clocks, clock increment and reordering) presents lower performance overhead than

	Protocol	Requests per second		No. of	Requests per second		No. of
		Average	Std. dev.	requests	Average	Std. dev.	requests
Eclipse Debugger	Normal	55,111.79	7,792.45	10^5	2.80	0.21	100
	Causal	55,172.60	5,764,97	10^5	3.39	0.30	100
AWED	Causal tags + clock inc.	56,079.85	5,983.75	10^5	234.77	5.07	10^5
	Invasive causality	53,045.19	10,223.90	10^5	237.61	7.58	10^5

Table 6.2: Debugging session without breakpoints (left half) and with a high-frequency breakpoint (right half).

the `Causal tags + clock increment` protocol. As this situation is not reproduced in the test with 80% writes, we can deduce that the network does not impose a large number of reorderings for low volumes of messages. Overall our new protocols do not impose a significant performance overhead (especially in the case of a large number of writes to the cache) compared to the standard JBoss Cache protocols.

6.5.3 Remote debugging vs. distributed debugging

In order to provide evidence that we have achieved the main objective set out in the motivation part, that is, that regular causal sequences improve on a per-host approach to debugging, we compare the performance of a remote debugging session with Eclipse and a distributed debugging session with AWED. To this end, we have again used the JBoss Cache benchmark framework. We first compare two debugging sessions, one with Eclipse and one with AWED, without breakpoints in order to measure the overhead of the frameworks. We then compare both debugging sessions in the presence of a high-frequency breakpoint (*i.e.*, reached and fired many times).

AWED runtime overhead vs. Eclipse remote debugging overhead. Table 6.2 (left part) compares the overhead of the debugging infrastructure posed by eclipse in a debugging session and the overhead posed by our AWED prototype. This test doesn't include any breakpoint, thus it only compares the overhead of the execution frameworks. The table shows small and comparable overhead for both frameworks. This is not surprising due to the fact that both frameworks are based on the Java agent technology and no breakpoints are evaluated. Note that the eclipse debugger is connected to the respective virtual machines, thus simulating a remote debugging session. AWED, by default, supports distribution.

As a last experiment we have compared the overhead of Eclipse and AWED in the presence of a high-frequency breakpoint: a breakpoint in the method `invoke` of the interceptor class `ReplicationInterceptor`. Table 6.2 (right part) shows the behavior of the Eclipse debugger attached to four nodes running the JBoss Cache framework and the behavior of AWED breakpoints under such conditions. In table 6.2 the protocol configuration labeled as *invasive causality* implies that the application being debugged has been invasively modified with an adapter for causality, thus AWED system can predicate over application's own messages. Using the Eclipse debugger we have executed the benchmarks first in JBoss Cache normal configuration and then with JBoss Cache using JGroups default `CAUSAL` protocol. The performance in these configurations is very bad and after several problems with memory overflow and unacceptable delays for the test we have reduced the number of requests to 100. On the other hand, the test of performance using the AWED framework are at least

seventy times faster and do not impose any restrictions in the conditions of the test. This is due to the fact that, even though the Eclipse debugger and AWED's dynamic framework use similar execution technology, AWED implements several optimization techniques and was designed with distribution in mind [BNSVV06]. Our approach thus scales much better than the discussed debugging methods using Eclipse.

Even though the evaluation presented here address only a partial set of features introduced by our approach it already gives important data to asses such features and gives insights for future work. Concretely, we have shown that current debugging practices in distributed middleware can benefit form such granular control over event ordering. We also show that current implementation strategies for distributed debugging can be improved by aspect techniques for distributions as dynamic weaving, finner grained language control, distributed deployment, and no centralized architectures with support for causality. Furthermore, the combination of finite state machines definitions with causal and concurrent constructs allow the construction complex and interesting patterns, *e.g.*, specifying concurrent events of interest within a time frame (delimited by specific events). However, the need for an graphical tool for dedicated debugging is evident, in order to realize systematic studies over different sets of applications. In particular, future work should focus in the evaluation of systematic ordering scenarios with several untracked and unresolved bugs. Additionally, even though we test the causal connectors specified for JGroups we need to evaluate the RMI connector and develop new ones, for example in the context of the different languages supported in ActiveMQ.

6.6 Discussion

In this chapter, we have argued for the use of programming abstractions as expressive support for the debugging and testing of distributed middleware, in particular for the definition of sophisticated relationships between distributed events and the recognition of event sequences in the presence of non-deterministic executions. We have presented a corresponding aspect-based language and implementation support that introduces causal event sequences into AWED, an aspect system for the dynamic manipulation of distributed systems. We have validated our approach in the context of Java-based middleware, in particular for the debugging and unit testing of a JBoss Cache and Apache's ActiveMQ. This evaluation has shown that our implementation has reasonable overhead and that our approach significantly improves on the use of debuggers, such as Eclipse, that are based on the manual coordination of per-host debugging sessions.

This work paves the way for several leads of future work. On a conceptual level, more flexible abstractions to define relationships that mix events that partially are causally ordered and partially are not are of foremost interest. Furthermore, exploring the use of our abstractions in other application domains, such as grid infrastructures, should be explored.

Chapter 7

Conclusion and future work

Crosscutting has been identified as a fundamental problem in the implementation of distributed software systems. However, few approaches for the modularization of crosscutting concerns in distributed applications have been proposed. In this thesis, we have investigated the problem of crosscutting due to the concurrent and distributed nature in large-scale applications, and developed better means to modularize such crosscutting concerns. In particular we have proposed the following contributions:

Crosscutting in large distributed applications. As part of this thesis work, we have studied the complexity of distributed software due to crosscutting concerns in object oriented implementations. In particular, we have investigated how code for distribution and concurrency is scattered and tangled in distributed object oriented middleware, such as JBoss Cache. The study has shown that even after several cycles of re-engineering and refactoring that crosscutting was resistant to modularization efforts using non aspect-oriented mechanisms. This resistance has provided strong evidence of the need for aspects with explicit distribution.

Language for distributed aspects. We have provided a model, corresponding programming language and implementation for aspects with explicit distribution (the AWED model and system) that incorporates several new mechanisms for remote pointcuts, remote advice, and distributed aspects as the three main abstractions for the modularization of crosscutting concerns. First, remote pointcuts enable matching of join points on remote hosts and include support for remote calls and distributed control flow constructs. AWED's pointcut model also supports the definition and matching of remote regular sequences of events. These sequence pointcuts have been further extended to support fine-grained control over distributed message ordering. We have integrated, in particular, support for the causal ordering of messages. Second, AWED includes remote advice that permits advice to be executed in a synchronous and asynchronous fashion with advice, transparent futures for synchronization of asynchronous executions, as well as support for different parameter passing modes, transparent remote object references, and customized policies for filtering and ordering of remote executions. Third, the model for distributed aspects provides a class-like abstraction to encapsulate pointcuts, and advice definitions to deal with remote deployment, instantiation, and data sharing among hosts. Finally, orthogonally to these main features AWED supports groups of host to be used in and manipulated as part of distributed aspects.

Invasive patterns for distributed programming .

We have explored the notion of distributed aspects in order to define new program-level abstractions for distributed programming. In particular, we have investigated how implicit interactions among the major runtime entities of distributed applications can be declaratively and concisely defined. we have shown that such implicit interactions can lead to intricate tangled and scattered code that is very difficult to understand and maintain. As a first step towards a general solution to this problem, we have proposed invasive patterns as a new mechanism for distributed programming languages that extend standard parallel communication and computation patterns to modularize the crosscutting code governing such interactions. We have implemented invasive patterns by a transformation into AWED and presented several examples of how it can be used to implement those complex distributed protocols that are commonly hidden in distributed applications when standard programming and implementation techniques are used.

Causal distributed aspects. Based on AWED's support for distributed sequences of events we have introduced support for causality between events in distributed aspects. Concretely, we have investigated how debugging and testing tasks in distributed applications require monitoring of intricate relationships between execution events occurring in different hosts. Due to the lack of support in current OO languages, such relationships are frequently defined implicitly and very difficult to observe and manipulate. In order to address this problem, we have extended sequence pointcuts with causality guards. Hence, guarded finite automata may be defined that may predicate over the causal relationship of events, thus identifying, *e.g.*, two events to occur concurrently or to be causally related. We have implemented this model by exploiting logical clocks (specifically vector clocks), and developed several techniques to adapt legacy applications to support causal predicates transparently. Finally, we have applied this technique successfully to the debugging and tests of different distributed middleware.

Application to large scale middleware and applications. To validate the notions, methods and techniques developed as part of this we have realized several experiments over industrial medium to large-scale applications. First, we have applied AWED to extend and refactor replicated caches, and to manage service composition in a service-oriented framework. Second, we have applied invasive patterns to implement transactional replicated protocols over distributed caches, and to define and implement a check-pointing algorithm over grid applications. Finally, we have used causal aspects to debug distributed applications like message brokers and compared our result with real mainstream debuggers. These experiments have given qualitative and quantitative information about the benefits of the proposed models.

7.1 Future work and perspectives

The research reported on in this thesis paves the way for different leads of future work in the short and medium term, as well as perspectives for more fundamental results in the long term. As to future work, our results should be beneficial for future work investigating concepts and semantics of aspect and distributed programming languages, corresponding implementation techniques and tool support. An interesting long term perspective consists in a general notion of architectural programming.

Concepts and semantics for aspect and distributed programming languages. AWED and invasive patterns have been defined informally and validated by concrete implementations and experiments. We have already proposed a first step towards a formal models for AWED and invasive patterns [BNDNS08]. This work presented two semantics based on labeled transitions systems (LTS), one for AWED and one for invasive patterns. Using these semantics we prove liveness and safety properties. However, the formal definition of the semantics and analysis of properties of AWED, more generally, aspectual concepts for concurrent and distributed programs such as invasive patterns represents a fundamental, diverse and highly interesting open issue.

Implementation of new abstractions. The implementation of AWED allows the experimentation of several semantic variants including full propagation of events, controlled propagation (*i.e.*, by means of advice chains), or control flow predicates over legacy RMI applications. Experimentation with such variants and its interactions, and experimentation with optimized constructs for finite-sate pointcuts constitute a rich source of ideas for future research. Furthermore, a recent shift in AOP research provides more imperative control over aspect constructs and mechanisms, especially as far as scoping, instantiation, and deployment of distributed aspects is concerned.

Tool development. The implementation of development and re-engineering tools for aspects with explicit distribution and invasive patterns, the implementation of optimized distributed debuggers using causal patterns, and the exploration the benefits of the state-machine approach [Sch90, Lam78] (*i.e.*, finite-state pointcuts combined with total and causal ordering of message to address replication and determinism) also opens a whole range of interesting leads for future work.

Architectural programming. We have shown that the definition of complex distributed protocols are often hidden in sequential object oriented implementations. Such protocols can be seen as embodying the relations that are essential to the definition of the runtime architectures of distributed applications. We have shown that there is a clear mismatch between the definition of such abstract architectures and their implementation. Invasive patterns are a first step towards closing this gap. Concretely, invasive patterns present a model that provides concrete language abstractions to define communication protocols among the different elements of a distributed architecture. As we have shown for JBoss Cache, they enable complex distributed systems with heterogeneous patterns of computations and synchronization be implemented in a way that allow to declaratively define their abstract runtime architectures on the program level. Generalizing this idea, such architectural programming would unite the benefits of declarative and concise architectural descriptions with the precision of concrete implementations.

Bibliography

[A+05] C. Allan et al. Adding trace matching with free variables to AspectJ. In R. P. Gabriel, editor, *Proc. of OOPSLA'05*. ACM Press, oct 2005.

[ACEF04] Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, sep 2004.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 202–211, New York, June 13–15 2004. ACM Press.

[AMC+03] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., Mountain View, CA, USA, 2003.

[And01] James H. Anderson. Lamport on mutual exclusion: 27 years of planting seeds. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 3–12, New York, NY, USA, 2001. ACM Press.

[asp08] AspectJ home page. `http://www.eclipse.org/aspectj`, 2008.

[AWB+94] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In *ECOOP '93: Proceedings of the Workshop on Object-Based Distributed Programming*, pages 152–184, London, UK, 1994. Springer-Verlag.

[AWE08] Awed home page. `http://www.emn.fr/x-info/awed`, 2008.

[Ban02] Bela Ban. JGroups, a toolkit for reliable multicast communication. http://www.jgroups.org/, 2002.

[Bir94] Ken Birman. A response to cheriton and skeen's criticism of causal and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 28(1):11–21, 1994.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd., Chichester, UK, 1996.

[BNDNS08] Luis Daniel Benavides Navarro, Rémi Douence, Angel Nuñez, and Mario Südholt. LTS-based semantics and property analysis of distributed aspects and invasive patterns. In *3rd International Workshop on Aspects, Dependencies and Interactions at the 22nd European Conference on Object-Oriented Programming ECOOP'08*, July 2008.

[BNDS08a] Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt. Aspect-based patterns for grid programming. In *In proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing*, Campo Grande,MS, Brasil, October 2008. IEEE Computer Society.

[BNDS08b] Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *In proceedings of the ACM-IFIP-USENIX 9th International Middleware Conference*, Leuven, Belgium, December 2008. Springer-Verlag. In press.

[BNSDM07a] Luis Daniel Benavides Navarro, Mario Südholt, Rémi Douence, and Jean-Marc Menaud. Invasive patterns: aspect-based adaptation of distributed applications. In *4th International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'07) at the 21st European Conference on Object-Oriented Programming ECOOP'07*, July 2007.

[BNSDM07b] Luis Daniel Benavides Navarro, Mario Südholt, Rémi Douence, and Jean-Marc Menaud. Invasive patterns for distributed programs. In *Proc. of the 9th International Symposium on Distributed Objects, Middleware, and Applications (DOA'07)*, LNCS. Springer Verlag, November 2007. ISSN: 0302-9743.

[BNSSS07] Luis Daniel Benavides Navarro, Christa Schwanninger, Robert Sobotzik, and Mario Südholt. Atoll: Aspect-oriented toll system. In *6th Int. Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'06) at AOSD*, New York, NY, USA, 2007. ACM Digital Library.

[BNSV+06a] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, mar 2006.

[BNSV+06b] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. Research Report 5882, INRIA, mar 2006.

[BNSVV06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, and Bart Verheecke. Modularization of distributed web services using awed. In *Proc. of the th Int. Conf. on Distributed Objects and Applications (DOA'06*, LNCS 4276, pages 1449–1466. Springer Verlag, oct 2006.

[BO00] Greg Barish and Katia Obraczka. World wide web caching: Trends and techniques. *IEEE Communications Magazine*, May 2000.

[Bos98] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming (JOOP)*, 11(2):18–32, 1998.

[BW83] Peter Bates and Jack C. Wileden. An approach to high-level debugging of distributed systems: preliminary draft. *SIGSOFT Softw. Eng. Notes*, 8(4):107–111, 1983.

[CBMT96] Charron-Bost, Mattern, and Tel. Synchronous, asynchronous, and causally ordered communication. *DISTCOMP: Distributed Computing*, 9:173–91, sep 1996. Cited in Denis Caromel and Ludovic Henrio. A theory of Distributed Objects. Springer-Verlag, Berlin, Germany, page 9, 2005.

[CC04] Adrian M. Colyer and Andrew Clement. Large-scale AOSD for middleware. In Gail C. Murphy and Karl J. Lieberherr, editors, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004*, pages 56–65. ACM, 2004.

[CCM] Open management group (omg). corba components, version 3. `Document formal/02-06-65, June 2002.`

[CH05] Denis Caromel and Ludovic Henrio. *A theory of Distributed Objects*. Springer-Verlag, Berlin, Germany, 2005.

[Chi95] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA*, pages 285–299, 1995.

[Chi00] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336, 2000.

[Cho56] N. A. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT-2:113–124, 1956.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137 – 167, 1959.

[CK03] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59, New York, NY, USA, 2003. ACM.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[Con08] The OW2 Consortium. The JAC project. http://jac.objectweb.org/, 2008.

[Cor] IBM Corp. IBM Patterns for e-business Resources. `http://www-128.ibm.com/developerworks/patterns/library`.

[CS93] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, pages 44–57, 1993.

[CV07] Rick Chern and Kris De Volder. Debugging with control-flow breakpoints. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 96–106, New York, NY, USA, 2007. ACM.

[DFL+05] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *Proc. AOSD'05*. ACM Press, mar 2005.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of GPCE'02*, volume 2487 of *LLNCS*, pages 173–188. Springer-Verlag, oct 2002.

[DFS04] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. AOSD'04*. ACM Press, mar 2004.

[DFS05] Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[DJ04] Maja D'Hondt and Viviane Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 132–140, New York, NY, USA, 2004. ACM.

[DRGL+07] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason D. Campbell, Padmanabhan Pillai, and Todd C. Mowry. Distributed watchpoints: Debugging large multi-robot systems. *International Journal of Robotics Research*, 2007.

[dWF04] Rob F. Van der Wijngaart and Michael A. Frumkin. Evaluating the information power grid using the NAS grid benchmarks. In *High-Performance Grid Computing Workshop – HPGC, 18th International Parallel and Distributed Processing Symposium (18th IPDPS 2004), CD-ROM*, Santa Fe, New Mexico, USA, apr 2004. IEEE Computer Society.

[E+06] John Easton et al. *Patterns: Emerging Patterns for Enterprise Grids*. IBM Redbooks. IBM, jun 2006. `http://publib-b.boulder.ibm.com/abstracts/sg246 682.html`.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, jun 2003.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[ET08] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 168–179, New York, NY, USA, 2008. ACM.

[ETFD+08] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive scoping of distributed aspects. Submission, 2008.

[Eug07] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007.

[FB07] Bruno De Fraine and Mathieu Braem. Requirements for reusable aspect deployment. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition, 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, volume 4829 of *Lecture Notes in Computer Science*, pages 176–183. Springer, 2007.

[FCAB00] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[Fos01] Ian T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proc. of Euro-Par'01*, pages 1–4, London, UK, 2001. Springer Verlag.

[Fou08] Eclipse Foundation. Remote debugging in eclipse. http://www.eclipse.org, 2008.

[Fru01] R.F. Frumkin, M. Van der Wijngaart. Nas grid benchmarks: a tool for grid space exploration. *High Performance Distributed Computing*, pages 315–322, 2001.

[FVSB05] Bruno De Fraine, Wim Vanderperren, Davy Suvée, and Johan Brichau. Jumping aspects revisited. In Robert E. Filman, Michael Haupt, and Robert Hirschfeld, editors, *Dynamic Aspects Workshop*, pages 77–86, March 2005.

[FZ90] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pages 134–141, Washington, DC, 1990. IEEE Computer Society.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. ADDISON-WESLEY, 2005.

[Glo] Globus3 resource specification language (rsl). `http://www-unix.Globus.org/toolkit/ docs/3.2/ gram/ws/developer/ mjsrslschema.html`.

[GPB05] Mark Grechanik, Dewayne E. Perry, and Don Batory. Using aop to monitor and administer software for grid computing environments. In *Proc. of COMPSAC'05, Vol. 1*, pages 241–248. IEEE, 2005.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[gri] Gridgain, a computational grid framework. `http://www.gridgain.com/product.html`.

[GSF+05] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM.

[HBS+02] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. Technical report, Sun microsystem, 2002.

[HK90] Wenwey Hseush and Gail E. Kaiser. Modeling concurrency in parallel debugging. In *PPOPP*, pages 11–20, 1990.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of OOPSLA'02*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[ICP] ICP. *Internet Cache Protocol*. http://icp.ircache.net/.

[JBo08a] JBoss home page. `http://www.jboss.com/`, 2008.

[JBo08b] JBoss Cache home page. `http://labs.jboss.com/jbosscache`, 2008.

[JGr08] JGroups home page. `http://www.jgroups.org`, 2008.

[JVS06] Niels Joncheere, Wim Vanderperren, and Ragnhild Van Der Straeten. Requirements for a workflow system for grid service composition. In *Business Process Management Workshops*, pages 365–374, Berlin, Germany, 2006. Springer.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[KFRGC98] Marc-Olivier Killijian, Jean-Charles Fabre, Juan-Carlos Ruiz-Garcia, and Shigeru Chiba. A metaobject protocol for fault-tolerant CORBA applications. In *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems (17th SRDS'98)*, pages 127–134, West Lafayette, Indiana, USA, October 1998. IEEE Computer Society.

[KG02] Joerg Kienzle and Rachid Guerraoui. AOP: Does it make sense? the case of concurrency and failures. In *Proceedings ECOOP '2002*, volume 2374 of *LNCS*. Springer Verlag, 2002.

[KH+01] Gregor Kiczales, Erik Hilsdale, et al. An overview of AspectJ. In *Proc. of ECOOP'01*, LNCS 2072. Springer, jun 2001.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, jun 2001. Springer-Verlag.

[Kic96] Gregor Kiczales. Aspect oriented programming. In *Proc. of the Int. Workshop on Composability Issues in Object-Orientation (CIOO'96) at ECOOP*, jul 1996. Selected paper published by dpunkt press, Heidelberg, Germany.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference of Object-Oriented Programming, ECOOP97*, volume 1241/1997 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, May 1997. Springer Berlin / Heidelberg.

[KR79] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. In Antonio L. Furtado and Howard L. Morgan, editors, *Fifth International Conference on Very Large Data Bases, October 3-5, 1979, Rio de Janeiro, Brazil, Proceedings*, page 351. IEEE Computer Society, 1979.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Li03] Jun Li. Monitoring and characterization of component-based systems with global causality capture. In *23th International Conference on Distributed Computing Systems (23th ICDCS'2003)*, pages 422–, Providence, RI, may 2003. IEEE Computer Society.

[LJ06] Bert Lagaisse and Wouter Joosen. True and transparent distributed composition of aspect-components. In Maarten van Steen and Michi Henning, editors, *Middleware 2006, ACM/IFIP/USENIX 7th International Middleware Conference, Melbourne, Australia, November 27-December 1, 2006, Proceedings*, volume 4290 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 2006.

[LS76] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, 1976. Cited in Jim Gray. Notes on Data Base Operating Systems. Springer-Verlag, London, UK, 1978.

[Mat88] Friedman Mattern. Virtual time and global states of distributed systems. In *Proceedings of the international Workshop on Parallel and distributed Algorithms*, Chateau de Bonas, France, October 1988.

[MK04] Giuliano Mega and Fabio Kon. Debugging distributed object applications with the eclipse platform. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 42–46, New York, NY, USA, 2004. ACM.

[MK07] Giuliano Mega and Fabio Kon. An eclipse-based tool for symbolic debugging of distributed object systems. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pages 648–666. Springer Berlin, 2007.

[MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[MS05] Giuseppe Milicia and Vladimiro Sassone. Jeeg: temporal constraints for the synchronization of concurrent objects: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):539–572, 2005.

[MWY91] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In P. America, editor, *Proceedings of the ECOOP '91 European Conference on Object-oriented Programming*, LNCS 512, pages 231–250, Geneva, Switzerland, July 1991. Springer-Verlag.

[Myc08] Sun Mycrosystems. Java naming and directory interface (JNDI). http://java.sun.com/products/jndi/, 2008.

[NS06] Dong Ha Nguyen and Mario Südholt. VPA-based aspects: Better support for AOP over protocols. In *SEFM*, pages 167–176. IEEE Computer Society, 2006.

[NST04] M. Nishizawa, S. Shiba, and M. Tatsubori. Remote pointcut - a language construct for distributed AOP. In *Proc. of AOSD'04*. ACM Press, 2004.

[PFFT02] M. Pinto, L. Fuentes, M.E. Fayad, and J.M. Troya. Separation of coordination in a dynamic aspect oriented framework. In *Proc. of AOSD'02*. ACM Press, 2002. short paper.

[PHK91] M. Krish Ponamgi, Wenwey Hseush, and Gail E. Kaiser. Debugging multithreaded programs with MPD. *IEEE Software*, 6(3):37–43, may 1991.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac: an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 34(12):1119–1148, 2004.

[PSH04] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA, 2004. ACM Press.

[RHH85] Jr. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[SDGS96] Stephen Siu, Mauricio De Simone, Dhrubajyoti Goswami, and Ajit Singh. Design patterns for parallel programming. In *Proc. of PDPTA'96*, volume I, pages 230–240. C.S.R.E.A. Press, aug 1996. University of Waterloo, Canada.

[sf08a] The Apache software foundation. Apache ActiveMQ is an open source message broker. http://activemq.apache.org/, 2008.

[sf08b] The Apache software foundation. Web site of the apache software foundation. http://www.apache.org/, 2008.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ . In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of OOPSLA'02*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 174–190, New York, nov 2002. ACM Press.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, 7(3):149–174, 1994.

[Sof08] Allinea Software. Distributed debugging tool. http://www.allinea.com/, 2008.

[Sou95] Jiri Soukup. *Implementing patterns*, pages 395–412. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[spr08] The spring framework. http://springframework.org/, 2008.

[SS98] L.M. Silva and J.G. Silva. System-level versus user-defined checkpointing. *Reliable Distributed Systems, Proc. of the 17th IEEE Symposium on*, pages 68–74, Oct 1998.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley and Sons Ltd., Chichester, UK, 2000.

[Süd07] Mario Südholt. Towards expressive, well-founded and correct Aspect-Oriented Programming. Habilitation thesis, University of Nantes, jul 2007. `http://www.emn.fr/sudholt/hdr/thesis.pdf`.

[SVAR04] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In *ICSE*, pages 418–427. IEEE Computer Society, 2004.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[Sü07] Mario Südholt. *Towards expressive, well-founded and correct Aspect-Oriented Programming*. PhD thesis, Habilitation (HDR) thesis, University of Nantes, July 2007.

[TG98] Ashis Tarafdar and Vijay K. Garg. Predicate control for active debugging of distributed programs. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, page 763, Washington, DC, USA, 1998. IEEE Computer Society.

[TN05] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In Robert Glück and Mike Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia, September 2005. Springer-Verlag.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices , 38(11).

[TS+03] Kai Tan, Duane Szafron, et al. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proc. of PPOPP'2003*, jun 2003.

[TT06] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, LNCS 4025. Springer, 2006.

[Tur36] A. Turing. On computable numbers with an application to the entscheidungs problem. *Proc. London Mathematical Society*, 2(42):230–265, 1936.

[VL97] Cristina Isabel Vidiera Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Dec 1997.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Commun. ACM*, 39(4):76–83, 1996.

[VS04] Wim Vanderperren and Davy Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In Robert Filman, Michael Haupt, Katharina Mehner, and Mira Mezini, editors, *DAW: Dynamic Aspects Workshop at AOSD'04*, pages 120–134, March 2004.

[VSCDF05] W. Vanderperren, D. Suvee, M. A. Cibran, and B. De Fraine. Stateful aspects in JAsCo. In *Proc. of Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer-Verlag, apr 2005.

[WV04] Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Foundations of Software Engineering (FSE)*, pages 159–169. ACM, October 2004.